Lecture Notes in Artificial Intelligence 2062

Subseries of Lecture Notes in Computer Science
Edited by J. G. Carbonell and J. Siekmann

Lecture Notes in Computer Science
Edited by G. Goos, J. Hartmanis and J. van Leeuwen

Springer
Berlin
Heidelberg
New York
Barcelona
Hong Kong
London
Milan
Paris
Singapore
Tokyo

Alexander Nareyek

Constraint-Based Agents

An Architecture for Constraint-Based Modeling and Local-Search-Based Reasoning for Planning and Scheduling in Open and Dynamic Worlds

Springer

Series Editors

Jaime G. Carbonell,Carnegie Mellon University, Pittsburgh, PA, USA
Jörg Siekmann, University of Saarland, Saarbrücken, Germany

Author

Alexander Nareyek
GMD FIRST
Kekuléstr. 7, 12489 Berlin, Germany
E-mail: alex@ai-center.com

Cataloging-in-Publication Data applied for

Die Deutsche Bibliothek - CIP-Einheitsaufnahme

Nareyek, Alexander:
Constraint based agents : an architecture for constraint based
modeling and local search based reasoning for planning and scheduling
in open and dynamics worlds / Alexander Nareyek. - Berlin ; Heidelberg ;
New York ; Barcelona ; Hong Kong ; London ; Milan ; Paris ;
Singapore ; Tokyo : Springer, 2001
(Lecture notes in computer science ; Vol. 2062 : Lecture notes in
artificial intelligence)
ISBN 3-540-42258-7

D. 83

CR Subject Classification (1998): I.2, F.2.2, C.2.4, E.5

ISBN 3-540-42258-7 Springer-Verlag Berlin Heidelberg New York

Springer-Verlag Berlin Heidelberg New York
a member of BertelsmannSpringer Science+Business Media GmbH

http://www.springer.de

© Springer-Verlag Berlin Heidelberg 2001
Printed in Germany

Typesetting: Camera-ready by author, data conversion by PTP Berlin, Stefan Sossna
Printed on acid-free paper SPIN 10781616 06/3142 5 4 3 2 1 0

Foreword

Planning, constraints, agents, and interactive gaming in a dynamic environment – four areas that make for an exciting research challenge. These are the areas chosen by Alexander Nareyek for his Ph.D. thesis work, his work on the Excalibur agent's planner, and the basis for the materials in this book. Future intelligent systems will work with other intelligent systems (people and machines) in open, dynamic, and unpredictable environments, and the ability to interleave sensing, planning, and execution for an agent will become an increasingly important topic.

Representing and reasoning about the constraints on activity or behavior in a domain is a powerful paradigm, which is useful in so many ways. It is especially useful when working in dynamic environments alongside other human and computer agents all with their own tasks, capabilities, and skills, which may be teamed and cooperatively deployed, or which may oppose one another. Such environments can only be partially modeled and there is much uncertainty to be coped with. Interactive gaming is a fertile area for finding ways in which to develop, test, and even deploy some of these powerful ideas.

Powerful and extendible ways to perform local reasoning and constraint management while retaining the global perspective to guide the search allows for a whole range of techniques which can be adapted to and scaled up to realistic applications. The methods can allow for the use of special purpose reasoners or constraint solvers for some of the constraint types. Yet the methodology allows for a coherent overview to be maintained.

I had the pleasure of working with Alex as part of his organizing team for the Workshop on Constraints and Planning at the National Conference of the American Association of Artificial Intelligence in 2000 (AAAI-2000). This workshop brought together a number of researchers who are seeking to mate planning technology developed in AI over some years, with the most powerful new techniques emerging from constraint satisfaction approaches. Alex's own work, reported in this book, is a key contribution to this important symbiosis. It could lead to much more realistic ways to employ planning and constraint satisfaction methods together for a range of tasks.

The work in this thesis also offers a basis to support multiple agents working in a dynamic mixed-initiative manner. This will require the representation of shared knowledge about tasks, behaviors, and plans, and local reasoning with this knowledge to allow for agents to perform their roles. Constraints on behavior offer a basis for such shared knowledge to support inter-agent activity.

The themes described in this thesis are at the very heart of AI approaches to planning, scheduling, and inter-agent communication and cooperation – the very stuff of Intelligent Systems.

Prof. Austin Tate, University of Edinburgh

Preface

Autonomous **agents** have become a key research area in recent years. The basic agent concept incorporates proactive autonomous units with goal-directed behavior and communication capabilities. These properties are becoming increasingly important, given the ongoing automation of human work. Users do not have to specify *the way* something is to be executed but rather *the goal* that is to be achieved. A reasonable way to pursue these goals must be found by the agents themselves. The agents do not have to act individually but can cooperate and perform coordinated group actions. Applications in electronic commerce, industrial process control, and the military sector are only the precursors of numerous forthcoming applications.

This book focuses on autonomous agents that can act in a **goal-directed** manner under **real-time constraints** and with **incomplete knowledge**, being situated in a **dynamic environment** where **resources may be restricted**. The real-time requirement means that an agent must be able to react within a small upper bound of response time, like milli- or microseconds. This is very important in dynamic environments, in which the agent must take external events into account. In addition, the agent's knowledge may be incomplete in many ways. Our main concern will not be with nondeterminism, i.e., different possible outcomes of actions; instead we will focus on the system's ability to handle a partially observable environment, i.e., where the closed-world assumption does not hold and a potentially infinite number of objects exist. Furthermore, the agent's actions may be constrained with respect to various resources, like food, energy, or money, and an agent may have optimization goals related to these resources. To satisfy these high requirements, this book enhances and combines paradigms like **planning**, **constraint programming,** and **local search**.

The application domain of this work is **computer games**, which fit the problem context very well since most of them are played in real time and provide a highly interactive environment where environmental properties are constantly changing. Low-level environment-recognition problems – like the processing of visual information and the spectrographic analysis of a noise – can be ignored, given the high-level environment information from the game engine, and the domain is variable enough to model all kinds of problem examples. In addition, these techniques are in great demand by the computer-games industry. This book therefore represents a useful combination of scientific contributions and application demands.

Chapter 1 elaborates the research subject, describes previous approaches, compares them, and draws conclusions for the techniques applied. The main contributions of this book are presented in Chapters 2 to 4.

To realize a declaratively formulated and efficiently executable search for an agent's behavior plan, the framework of constraint programming is applied as the basic paradigm throughout the book. Since dynamics and real-time computation must be supported, a combination of constraint programming with local search is developed in Chapter 2. An inclusion of domain-dependent knowledge to guide and accelerate search is achieved by so-called global constraints. These techniques form the basis for the real-time computation of an agent's behavior, while preserving the properties of declarativeness and variable applicability.

The problem of reasoning about an arbitrary number of objects in an agent's world and about an arbitrary structure of an agent's plan is tackled in Chapter 3. The framework of constraint programming is not normally designed for this purpose and is therefore enhanced by adding the ability to handle problems in which the search for a valid structure is part of the search process. The concept is combined with the local-search approach treated in Chapter 2. The techniques described in Chapter 3 establish the basis for ensuring that the search for an agent's plan can be carried out free from restrictions, like considering only a predefined number of objects in the world, and make it possible to guide the search toward interesting features, such as the optimization of resource-related properties.

The concepts and techniques treated in Chapters 2 and 3 were not specifically designed for the planning domain only and are applicable to a whole range of other domains such as configuration and design. Chapter 4 applies them to an agent's behavior planning, introducing the agent's planning model. The model focuses on resources and also allows a limited handling of an agent's partial knowledge.

The interplay of all parts – the realization of an autonomous agent – is demonstrated in Chapter 5. Some general solving heuristics are presented and applied to the Orc Quest problem and variations of the Logistics Domain. Real-time computation, a plan property's optimization, and the handling of dynamics are demonstrated.

The material presented here is part of the Excalibur project[1] and is largely based on the publications [127] to [136]. More information on the Excalibur project is available at:

`http://www.ai-center.com/projects/excalibur/`

[1] Please note that there is no relation to Brian Drabble's Excalibur planner [43].

Acknowledgments

I should like, first of all, to thank Professor Stefan Jähnichen for his general help and his support in context with the postgraduate course Communication-Based Systems. My thanks also go to the other professors involved in the postgraduate course and to the German Research Foundation (DFG) for providing the enhanced Ph.D. scholarship. The heated discussions in the postgraduate course were most helpful, the course's heterogeneous composition often forcing me to look at my work from another perspective.

My work greatly benefited from the collaboration with the planning & optimization department at the GMD-FIRST institute. The head of this department, Professor Ulrich Geske, supported me in every possible way, and I had many fruitful discussions with members of the department, e.g., Ulrich John and Dr. Armin Wolf – to mention but a few. The GMD-FIRST has a very pleasant work atmosphere, and I enjoyed a great deal of freedom in my research. I was also able to use the institute's very fine infrastructure and was provided with funding to attend conferences. These conference visits proved a highly positive experience, and I would advise every Ph.D. student to take advantage of such opportunities.

The detailed documentation of my work in the Internet drew an unexpectedly large resonance, and I wish to thank everybody that offered comments and suggestions on the Excalibur project's presentation.

Many of the reviewers of the papers on which this thesis is based offered valuable suggestions. In fact, in a sudden burst of opportunism, I decided to confine the rather humorous style to Sect. 1.3.2 because a reviewer of a paper that included an earlier version of this section "hated" the "writing style", and I don't want to put off solemn characters.

I am indebted to a great many other people for helping me while I was writing this thesis, too many to list them all. However, I wish to especially thank Dr. Gabriele Taentzer for help on algebraic graph grammars; Professor Eugene C. Freuder and Richard Wallace for providing me with the DIMACS travel grant, and Excalibur's commercial partners for their support of the project, which was closely related to my Ph.D. studies. My special thanks go to Andreas Graf of Cross Platform Research Germany, Nicolas A. L. Cosio of NICOSIO, and Johann Lotter of Conitec Datensysteme GmbH. In addition, many people helped me with practical and administrative matters such as organizing workshops related to the subject of my thesis. Many thanks to all of these!

Finally, I wish to express my gratitude to Professor Steve Chien and Professor Bernhard Nebel for acting as external reviewers, to Philip Bacon for polishing up my English, and to our system administrator Roger Holst for supporting my numerous test runs.

March 2001 Alexander Nareyek

Table of Contents

1. Introduction

The following sections describe various approaches for dealing with important aspects of the application domain, comparing them, and discussing which are suitable for our specific requirements. A summary is given in Sect. 1.6, leading on to a discussion on the techniques that must be developed or enhanced. Internet resources for the sections' topics are listed in Appendix A.

Section 1.1 begins by introducing the computer-games application domain. This leads on to the concept of agents, which is discussed in Sect. 1.2. A key property of an agent is its planning capability. Techniques for planning are examined in Sect. 1.3. The planning process requires the application of sophisticated search mechanisms, which are reviewed in Sect. 1.4 and 1.5.

1.1 Artificial Intelligence for Computer Games

The use of game applications has a long tradition in artificial intelligence (AI). Games provide high variability and scalability for problem definitions, are processed in a restricted domain and the results are generally easy to evaluate. But there is also a great deal of interest on the commercial side, the "AI inside" feature being a high-priority task [29, 32, 171] in the fast-growing, multi-billion-dollar electronic gaming market (the revenue from PC games software alone is already as big as that of box office movies [84]).

Many "traditional" games, such as card/board/puzzle like Go-Moku [6] and the Nine Men's Morris [62], have recently been solved by AI techniques. Deep Blue's victory over Kasparov was another milestone event here. However, the success of many game programs can mainly be attributed to the vast increase in computing power – many researchers using exhaustive-search rather than knowledge-based methods [140]. While these applications bear impressive witness to advances in hardware, no noticeable scientific contribution has been made and such advances are of little help in solving much more complex real-world problems. Results like *Go-Moku is a win for Black* or *random instances of Rubik's Cube can be solved optimally* are not really applicable to other areas. Of the techniques used in this field of research, A* [82] (an improved version of Dijkstra's shortest-path algorithm [40]) and its variants/extensions are practically the only ones employed in "modern" computer games – like action, adventure, role-playing, strategy, simulation and sports games (see also [192]).

Such games pose problems for AI that are infinitely more complex than those of traditional games. Modern computer games are usually played in real time, allow very complex player interaction and provide a rich and dynamic virtual environment. Techniques from the AI fields of autonomous agents, planning, scheduling, robotics and learning would appear to be much more important than those from traditional games.

AI techniques can be applied to a variety of tasks in modern computer games. A game that uses probabilistic networks to predict the player's next move in order to speed up graphics may be on a high AI level. But although AI must not always be personified, the notion of artificial intelligence in computer games is primarily related to characters. These characters can be seen as *agents*, their properties perfectly fitting the AI agent concept.

But how does the player of a computer game perceive the intelligence of a game agent/character? This question is dealt with neatly in [90]. Important dimensions include physical characteristics, language cues, behaviors and social skills. Physical characteristics like attractiveness are more a matter for psychologists and visual artists (e.g., see [61]). Language skills are not normally needed by game agents and are ignored here too.

The most important question when judging an agent's intelligence is the goal-directed component. The standard procedure followed in today's computer games is to use predetermined behavior patterns (see Sect. 1.2.1). This is normally done using simple if-then rules. In more sophisticated approaches using neural networks, behavior becomes adaptive, but the purely reactive property has still not been overcome.

Many computer games circumvent the problem of applying sophisticated AI techniques by allowing computer-guided agents to cheat. But the credibility of an environment featuring cheating agents is very hard to ensure, given the constant growth of the complexity and variability in computer-game environments. Consider a situation in which a player destroys a communication vehicle in an enemy convoy in order to stop the enemy communicating with its headquarters. If the game cheats in order to avoid a realistic simulation of the characters' behavior, directly accessing the game's internal map information, the enemy's headquarters may nonetheless be aware of the player's subsequent attack on the convoy.

1.2 Agents

Wooldridge and Jennings [193] provide a useful starting point by defining autonomy, social ability, reactivity and proactiveness as essential properties of an agent. Agent research is a wide area covering a variety of topics. These include:

- *Distributed Problem Solving (DPS)*
  The agent concept can be used to simplify the solution of large problems by distributing them to a number of collaborating problem-solving units. DPS is not considered here because computer games' agents should normally act fully autonomously: Each agent has individual goals[1].
- *Multi-Agent Systems (MAS)*
  MAS research deals with appropriate ways of organizing agents. These include general organizational concepts, the distribution of management tasks, dynamic organizational changes like team formation and underlying communication mechanisms (see also Sect. 7.5).
- *Autonomous Agents*
  Research on autonomous agents is primarily concerned with the realization of a single agent. This includes topics like sensing, models of emotion, motivation, personality, and action selection and planning. This field is our main focus in the present book.

An agent has goals (stay alive, catch player's avatar, ...), can sense certain properties of its environment (see objects, hear noises, ...), and can execute specific actions (walk northward, eat apple, ...). There are some special senses and actions dedicated to communicating with other agents.

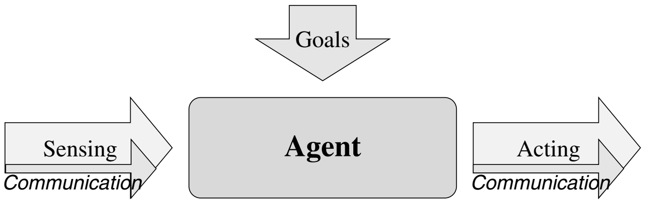


Fig. 1.1. Interaction with the Environment

The following sections classify different agent architectures according to their trade-off between computation time and the realization of sophisticated goal-directed behavior.

1.2.1 Reactive Agents

Reactive agents work in a hard-wired stimulus-response manner. Systems like Joseph Weizenbaum's Eliza [188] and Agre and Chapman's Pengi [2] are examples of this kind of approach. For certain sensor information, a specific action is executed. This can be implemented by simple if-then rules.

[1] Superior common goals, like catching the player's avatar, can be realized by setting them as individual goals for each agent (see [173] for more sophisticated approaches).

The agent's goals are only implicitly represented by the rules, and it is hard to ensure the desired behavior. Each and every situation must be considered in advance. For example, a situation in which a helicopter is to follow another helicopter can be realized by corresponding rules. One of the rules might look like this:

```
IF (leading_helicopter == left) THEN
  turn_left
ENDIF
```

But if the programmer fails to foresee all possible events, he may forget an additional rule designed to stop the pursuit if the leading helicopter crashes. Reactive systems in more complex environments often contain hundreds of rules, which makes it very costly to encode these systems and keep track of their behavior.

The nice thing about reactive agents is their ability to react very fast. But their reactive nature deprives them of the possibility of longer-term reasoning. The agent is doomed if a mere sequence of actions can cause a desired effect and one of the actions is different from what would normally be executed in the corresponding situation.

1.2.2 Triggering Agents

Triggering agents introduce internal states. Past information can thus be utilized by the rules, and sequences of actions can be executed to attain longer-term goals. A possible rule might look like this:

```
IF (distribution_mode) AND (leading_helicopter == left) THEN
  turn_right
  trigger_acceleration_mode
ENDIF
```

Popular Alife agent systems like CyberLife's Creatures [77], P.F. Magic's Virtual Petz [170] and Brooks' subsumption architecture [25] are examples of this category. Indeed, nearly all of today's computer games apply this approach, using finite state machines to implement it.

These agents can react as fast as reactive agents and also have the ability to attain longer-term goals. But they are still based on hard-wired rules and cannot react appropriately to situations that were not foreseen by the programmers or have not been previously learned by the agents (e.g., by neural networks).

1.2.3 Deliberative Agents

Deliberative agents constitute a fundamentally different approach. The goals and a world model containing information about the application requirements and consequences of actions are represented explicitly. An internal refinement-based planning system (see section on planning) uses the world model's information to build a plan that achieves the agent's goals. Planning systems are often identified with the agents themselves.

Deliberative agents have no problem attaining longer-term goals. Also, the encoding of all the special rules can be dispensed with because the planning system can establish goal-directed action plans on its own. When an agent is called to execute its next action, it applies an internal planning system:

```
IF (current_plan_is_not_applicable_anymore) THEN
  recompute_plan
ENDIF
execute_plan's_next_action
```

Even unforeseen situations can be handled in an appropriate manner, general reasoning methods being applied. The problem with deliberative agents is their lack of speed. Every time the situation is different from that anticipated by the agent's planning process, the plan must be recomputed. Computing plans can be very time-consuming, and considering real-time requirements in a complex environment is mostly out of the question.

1.2.4 Hybrid Agents

Hybrid agents such as the 3T robot architecture [19], the New Millennium Remote Agent [145] or the characters by Funge et al. [59] apply a traditional off-line deliberative planner for higher-level planning and leave decisions about minor refinement alternatives of single plan steps to a reactive component.

```
IF (current_plan-step_refinement_is_not_applicable_anymore) THEN

   WHILE (no_plan-step_refinement_is_possible) DO
     recompute_high-level_plan
   ENDWHILE
   use_hard-wired_rules_for_plan-step_refinement

ENDIF
execute_plan-step_refinement's_next_action
```

There is a clear boundary between higher-level planning and hard-wired reaction, the latter being fast while the former is still computed off-line. For complex and fast-changing environments like computer games, this approach is not appropriate because the off-line planning is still too slow and would – given enough computation time – come up with plans for situations that have already changed.

1.2.5 Anytime Agents

What we need is a continuous transition from reaction to planning. No matter how much the agent has already computed, there must always be a plan available. This can be achieved by improving the plan iteratively. When an agent is called to execute its next action, it improves its current plan until its computation time limit is reached and then executes the action:

```
WHILE (computation_time_available) DO
  improve_current_plan
ENDWHILE
execute_plan's_next_action
```

For short-term computation horizons, only very primitive plans (reactions) are available, longer computation times being used to improve and optimize the agent's plan. The more time is available for the agent's computations, the more intelligent the behavior will become. Furthermore, the iterative improvement enables the planning process to easily adapt the plan to changed or unexpected situations. This class of agents is very important for computer-games applications and will constitute the basic technology for our agents.

1.3 Planning

A crucial aspect of an agent is the way its behavior is determined, i.e., **what** has to be done **when**. If there is to be no restriction on reactive actions with predetermined behavior patterns, an underlying planning system is needed. A great deal of research has been done on planning, and a wide range of planning systems have been developed, e.g., STRIPS [55], UCPOP [146], PRODIGY [177], TLplan [11] and SHOP [138]. Many recent approaches are based on Graphplan [16] and SATPLAN [98, 99].

The basic planning problem is given by an initial world description, a partial description of the goal world, and a set of actions/operators that map a partial world description to another partial world description. A solution is a sequence of actions leading from the initial world description to the goal world description and is called a *plan*. The problem can be enriched by including further aspects, like temporal or uncertainty issues, or by requiring the optimization of certain properties[2].

1.3.1 Temporal Ontologies

Numerous different plan representations exist. There are two main ontologies: sequence-oriented and explicit timeline approaches. Figure 1.2 shows these paradigms.

[2] Note that the use of the term *planning* in AI is different from that expected by people in the operations research (OR) community (e.g., *scheduling*).

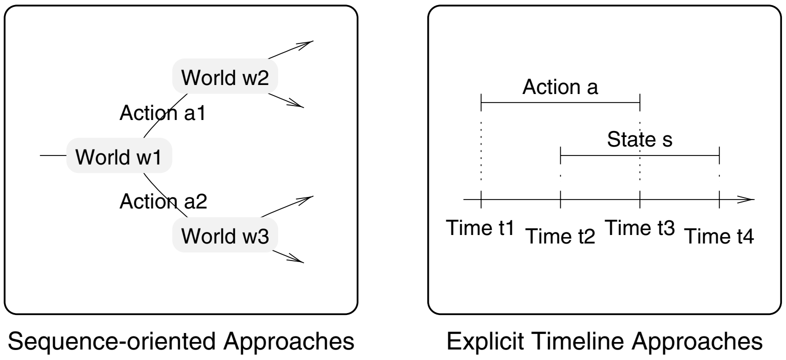


Fig. 1.2. Temporal Ontologies

Sequence-oriented approaches work on a branching time-point structure, in which world descriptions (sets of states) are linked by actions. In explicit timeline approaches, actions and states are related to a timeline. The sequence-oriented approaches' branching structure allows reasoning about multiple possible futures in a very direct way, whereas explicit timeline approaches focus on handling complex temporal relations.

Sequence-Oriented Representations. The most prominent representative of sequence-oriented representations is the STRIPS model by Fikes and Nilsson [55], which is based on the Situation Calculus by McCarthy and Hayes [118] and is applied in many planning systems.

In STRIPS, a single fact is represented by an atomic proposition, such as `IS_OPEN(DOOR)`. A set of these facts establishes a world state.

A plan is a sequence of actions, an action being defined by a **precondition**, a **delete list** and an **add list**. The precondition is a formula, which must be valid in the current world state. To obtain the subsequent world state, the facts of the delete list must be deleted from the current world state, and the facts of the add list must be added. Here is an example of an action `MOVE(A, B, C)` that moves block `A` from the top of block `B` onto block `C`:

Action:	`MOVE(A, B, C)`
Precondition:	`CLEAR(A) & ON(A, B) & CLEAR(C)`
Add List:	{ `CLEAR(B), ON(A, C)` }
Delete List:	{ `ON(A, B), CLEAR(C)` }

Sequence-oriented representations provide a rich framework for the treatment of decision alternatives. But in our multi-agent domain with its dynamic environment and temporally complex actions, this would result in unmanageable complexity. Sequence-oriented representations have a branching point for each action decision of the agent. In a dynamic environment, the situation may change without the agent's activity. To capture these changes, each branching point would have to include all possible changes of the environment that might affect the agent's behavior. As computer game agents are commonly required to be highly interactive with their environment, this entails a vast combinatorial explosion[3]. The incorporation of a richer temporal annotation (actions have a duration; simultaneity; synergies) is even more fatal. There must be branching points for each possible timestep providing for all possible changes in the environment that might affect the agent's behavior – a desperate approach.

The success of sequence-oriented representations like STRIPS is a result of assumptions negating external events and a richer temporal annotation. It must be said, though, that there are several extensions of STRIPS-like representations designed to remove specific backdraws, e.g., an extension to concurrent interacting actions by Boutilier and Brafman [21].

Explicit Timeline Representations. The most prominent representative of explicit timeline representations is Allen's work on temporal intervals [3, 4]. Time intervals are the basic units, which are correlated by qualitative relations like BEFORE and OVERLAPS. A more efficient and flexible revision was presented by Freksa [57].

To represent quantitative information as well, metric information must be incorporated. Many planners with an explicit notion of time, e.g., zeno [147] and Descartes [92], use the approach of Malik and Binford [113], which employs linear inequalities to represent temporal (and spatial) information, and apply general constraint solvers. Other approaches use specific temporal management mechanisms, like the temporal constraint networks of Dechter, Meiri and Pearl [39] or the time map manager of Schrag, Boddy and Carciofini [163]. These are used in planning systems like HSTS [124].

An explicit timeline representation is a very general framework; it does not specify how to express relations between actions, events and states. This extension is often made in combination with a logic language, e.g., Allen's interval temporal logic [5].

For the planning of our agents, the disjunctive feature of sequence-oriented representations is of little interest because of its combinatorial explosion. The temporal expressiveness of explicit timeline approaches, on the other hand, is much more important.

1.3.2 Resources

Resources such as food, hit points or magical power are standard features of today's computer games. Computer-guided characters often have goals that are related to these resources and must take this into account in planning their behavior. Thus, besides temporal issues, it is vital that the expressiveness and the solving abilities of an agent's planning system are capable of handling resources as well as goals related to these resources.

[3] See Sect. 4.3.1 for a description of exhaustively branching planners.

Resource-based action planning systems have recently begun to mushroom – an indication that AI planning systems have finally started to tackle real-world problems. However, these systems still tend to neglect the resources. Conventional plan length is the primary optimization goal, resources being of only secondary importance. The inadequacy of these approaches is discussed in the following subsections. See Sect. 4.4 for a classification of specific resource-based planning systems.

The Orc Quest Example. Imagine you are a computer-guided Orc in a computer game[4]. Your interests are somewhat different from those of your philistine Orc friends. To get them a little more acquainted with the fine arts, you decided to surprise them with a ballet performance. As a highlight, five of those delicate humans are to play supporting roles. However, the uncivilized humans do not appreciate the charm of Orc-produced art and are extremely reluctant to participate. This makes it rather painful to catch a human and force him to take part in the performance, and much more painful to catch a whole group of them. On the other hand, going for a group saves time. This can be formalized as follows:

```
STATEVARS: Duration, Pain, Performers ∈ ℕ0

INIT: Duration = 0, Pain = 0, Performers = 0
GOAL: Performers ≥ 5

ACTION catch_only_one:
 Duration += 2, Pain += 1, Performers += 1

ACTION catch_a_group:
 Duration += 5, Pain += 4, Performers += 3
```

The Application of Planning Systems. Being an Orc, you lack sufficient brain power to solve the problem. You therefore apply a conventional AI planning system and come up with the following plan: `catch_a_group & catch_a_group`, which yields a plan duration of **10 hours**, an **8 on the pain scale** and **6 performers**. The plan attains your goal of catching at least 5 performers, but the other results look to you to be capable of improvement.

You realize that you have forgotten to apply an optimization criterion. You thus repeat the planning process, applying a state-of-the-art resource-based planning system that you request to minimize your pain, hoping dearly that none of your Orc friends will realize what a coward you are.

```
GOAL: Performers ≥ 5, MIN(Pain)
```

Strangely enough, the resource-based planning system delivers the same plan as the conventional planning system.

[4] For those who have not heard of Orcs before: "Orcs ... are the most famous of Tolkien's creatures. ... Orcs tend to be short, squat and bow-legged, with long arms, dark faces, squinty eyes, and long fangs. ... Orcs hate all things of beauty and love to kill and destroy." [115]

The Real Optimum. Shortly afterward, the first group of humans is caught and you decide to get them to verify the planning system's results. The three gawky simpletons believe your promise to release them if they help you. After a while, they present you with a plan of five sequential `catch_only_one` actions, which yields a plan duration of **10 hours**, a **5 on the pain scale** and **5 performers**. This answer worries you as it involves much less pain than the solution found by the planning system employed to minimize your pain. Is human AI technology merely a way of undermining the Orcs' preordained dominance in the fine arts?

A Question of Relevance. You call the humans to account for the behavior of the planning system. They reply that most resource-based planning systems consider only a limited number of actions, increasing this number only if they are unable to produce a *correct plan*, the primary optimization goal being *plan length*. If it is possible to produce a correct plan with a certain number of actions, optimization of other goal properties can begin and plans with a larger number of actions are no longer considered. For your planning problem, the plan `catch_a_group & catch_a_group` is the only correct plan with a *minimal number of two actions*, and thus also the optimum with respect to any secondary optimization criterion.

Taking the number of actions or the plan length as the primary optimization criterion seems a very academic approach to you and you wonder if it has any relevance. You notice that the humans are becoming increasingly uncomfortable when they start arguing that optimization of the plan's duration is what is needed in most cases. And the duration would be the same as the number of actions. You no longer trust these humans and demand that they draw a picture of the complete search space. The result does not improve the humans' situation because the plan with the shortest duration is different from the one involving the minimal number of actions (see Fig. 1.3).

Luckily for the group of humans, your goal is a plan involving the minimal amount of pain, and you reluctantly release the humans as promised because the plan involving the minimal amount of pain does not entail catching a whole group of humans. Somehow you feel dissatisfied at not having broken your promises.

The Nearby Wizard. A little bit later, it occurs to you that there is a wizard living nearby who can prepare a magic potion that lessens the pain by one on the pain scale. The drawback here is that the wizard is also very fond of humans because he is in continual need of test persons to improve his miserable skills in casting teleportation spells. Usually, then, he wants some humans in exchange for his magic potion, which he calls "ethanol".

You visit the wizard and explain your problem to him. He scratches his head, walks around for a while deep in thought and finally says: "All right, this is what we'll do: I'll give you 11 phials of potion for every 10 humans that you promise to bring to me." You are not really sure how many phials you should take and formalize this for your planning system as follows:

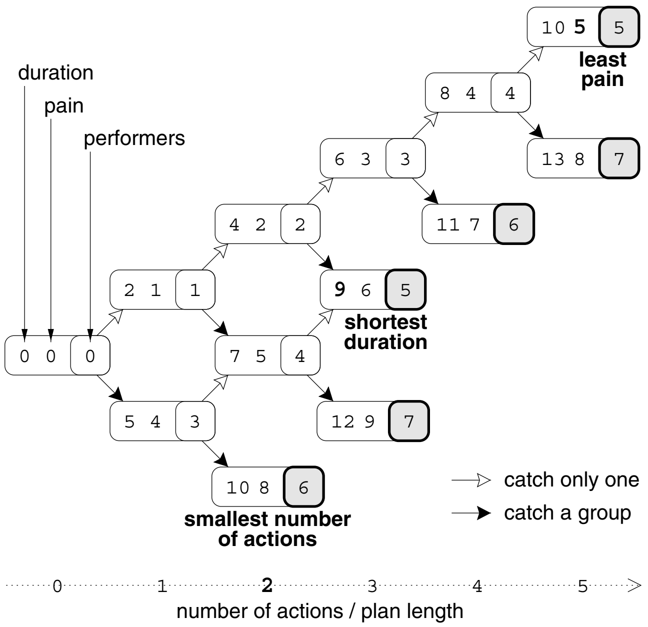


Fig. 1.3. The Search Space of the Orc Quest Example

```
ACTION deliver_humans:
 Duration += 1, Pain -= 11, Performers -= 10
```

Plan-Length Bounds. The optimization problem without the `deliver_humans` action could have been optimally solved by calculating an upper bound for the plan length and starting the planning system with the requirement to consider only plans with a length of this bound. A `no_action` action could be introduced to allow for plans that are shorter than the length allowed by the bound. The bound is easy to calculate because a plan's `Performers` and `Pain` increase strictly monotonically with the plan length, and after an application of 5 actions the goal condition for the `Performers` is met anyway, and the `Pain` can only get worse for longer plans.

However, this simple way of calculating an upper bound is no longer applicable with an action like `deliver_humans` that destroys the monotony. You think about the new problem for a bit, but it is far too complex for an Orc. Lacking a method to calculate the upper bound, you decide to run the planning system with some random bounds for the plan length. Starting with a bound of 6, the plan that consists of 5 times `catch_only_one` is returned. With a length of 7, you still get this solution. You decide to raise the bound to 15, and ... wait. After a long period, the solution is confirmed again. It seems to be the optimum, but you test a bound of 30 just to be sure, and ... wait. After a while, the planning system returns a different solution than before. Unfortunately, this solution is beyond your comprehension and you are gradually overcome by the unpleasant feeling that the suggested plan "`out of memory`" is just another feature of the system designed to mislead helpless Orcs.

The wizard is quite astonished when you tell him that his offer would not help you to improve your plan. You offer to demonstrate this with your planning system, but he wants to try his own. Shortly afterward, he presents you with a plan that entails no pain for you. The plan has a plan length of 60, and you wonder how his system was able to return a solution that quickly.

A Question of Efficiency. "The wrong approach taken by your planning system," the wizard starts to explain, "is due to its focusing on qualitative problems, like the ancient blocks-world domain. The only feature distinguishing solutions was the plan length, and so everyone wanted to optimize this. A systematic exploration of the search space along an increasing plan length is an obvious way of tackling this problem, and such approaches are still at the heart of today's planning systems, even if the problems now involve optimization criteria that do not change strictly monotonically with the plan length."

In response to your questioningly look, he continues: "Your planning system has a very strange way of exploring the search space." He draws a small figure on the ground (see Fig. 1.4).

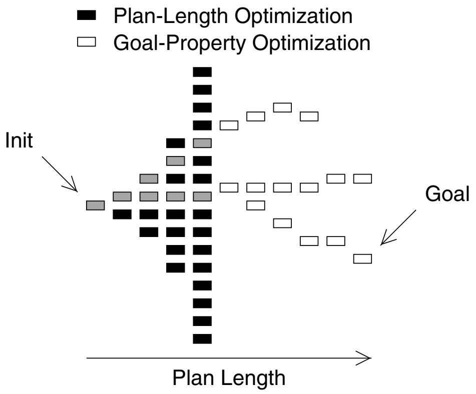


Fig. 1.4. Search-Space Exploration

"The plan-length optimization of your system," the wizard explains, "tries to verify that there is no valid solution for a certain plan length, and has to perform a complete search for this length before being able to increase the plan length. And even if you instruct it to begin with a certain length bound, it constructs the structures for the whole search space for this length, which is a pretty useless approach for non-toy problems.

As you have already realized, the plan-length criterion is a poor guide for the search because it has absolutely no relevance, so the search is rather like a random search. It is much more convenient to explore the search space without being bound to the plan length, as my planning system does. This allows us to perform a goal-property-driven search by integrating domain knowledge. Searching in an infinite search space means relying on powerful heuristics instead of being tied to a method of exploration that is determined by an irrelevant criterion."

Complexity. The wizard still sees a questioning look on the face in front of him, and he heaves a sigh: "Are you wondering about the infinite search space? Our planning problem is *undecidable* because it involves numbers with unpredictable bounds – you can read up on this in [53]."

Suddenly, while casting a light-the-candle spell the wizard's apprentice vanishes in a big ball of fire. He had always been eager to try out this spell, but was warned by the wizard about how dangerous it was. Now the apprentice had been eavesdropping while you were talking with the wizard and had concluded that the optimization of his life's duration is also an undecidable problem, and thus not worth considering.

The wizard shakes his head sadly: "Hardcore theorists! If a problem is undecidable, they hate to tackle it." Just to be sure that you will not fall victim to one of the wizard's teleporation spells, you quickly put in: "I'm not one of them!" "No," the wizard smiles, "Orcs don't tend to be theorists anyway. But perhaps this neglect of undecidable problems is the reason why for so long no one dared to tackle problems other than plan-length optimization."

Decision-Theoretic Planning. The wizard invites you to dinner, and you gladly accept. A little while later, you are sitting in a comfortable armchair with a jar of the wizard's ethanol potion in your hands. The time has come to ask where he got his planning system from. The wizard's potion seems to have a paralyzing effect on your tongue, but finally you manage to put your question in reasonably comprehensible terms.

The wizard takes a while to answer and gives you an earnest look. Then he raises his eyebrows meaningfully: "Have you heard of EXCALIBUR?" You are seized with a shivering fit because EXCALIBUR is a popular name for swords used by computer players' avatars that generally enjoy tyrannizing harmless Orcs.

"No, no," the wizard says appeasingly, "not the sword!" His voice takes on a vibrant tone: "Did you ever consider the possibility that you are part of a computer program and that your behavior is largely guided by a planning system?" He pauses briefly to let you reflect on all this question's implications. "Well," he continues, casting a thunderbolt to conjure up a more threatening atmosphere, "what I mean is the EXCALIBUR Agent's Planning System!" He lowers his voice: "We are the *agents*. You understand?"

This sounds like one of those popular conspiracy theories. You are, however, not particularly well versed in philosophy and already have difficulty explaining why there are now two wizards and why all of the equipment in the room is swaying curiously. “Please put off describing the system for now,” you say, putting an end to this rather dubious discussion. “If resources are such a vital component, why aren’t there other planning systems like the Excalibur agent’s planning system?”

“Well,” the wizards say, “I must admit that there have, of course, also been other approaches at trying to focus on optimizing things other than plan length. These are mostly subsumed under the term *decision-theoretic planning*. But most of the research in this area aims to optimize probabilistic issues for reasoning under conditions of uncertainty[5]. Although goals like maximizing probability to attain a goal can also be interpreted as resource-related optimization (the probability being the resource to be maximized), probability is usually considered to have only monotonical features, i.e., a plan cannot have a higher probability of attaining the goal than a subplan that also attains the goal. Your problem involving an action like using a magic potion has no counterpart here.

But there are also resource-oriented planning systems that are able to handle more complex problems, e.g., Williamson and Hanks’ Pyrrhus planning system [190] and Ephrati, Pollack and Milshtein’s A*-based approach [52]. The difference between these and the Excalibur agent’s planning system is the conceptual approach.”

“Sorry, did you say *A**? It’s funny that you should mention this term because it sometimes comes mysteriously to mind – usually when I think about how to get to a certain place. Do you know what it means?”

“Well, all it means is that the Excalibur agent’s planning system has specialized heuristics for specific subtasks. Don’t worry about it! But – A* brings me directly to the point. The resource-oriented planning systems mentioned above construct plans in a stepwise manner, e.g., action by action, and if they fail, they go back to an earlier partial plan and try to elaborate this. In order to choose a partial plan for an elaboration – called *refinement* – an upper bound of the quality is computed for each partial plan. The plan that yields the highest upper bound is chosen for elaboration. Sounds like a great concept, doesn’t?”

“Probably not – since you ask!” you reply and grunt loudly, impressed by your own cleverness. The wizards give you a rather disconcerted look, but quickly collect themselves and continue: “Er, you’re right, the concept is great for things like path finding, for which it is easy to determine bounds because the actions can only add values. But for complex planning problems it is not normally possible to calculate bounds on the quality value for a partial plan. We spoke about the problems with bounds before. Does a single action like building a trap for someone puts bounds on the total fun you will have?”

“Definitely not! Things can actually get worse!” you answer bad-temperedly, “If I don’t take precautions, the person trapped may be a player’s

[5] See Sect. 4.3.1 for a description of probabilistic planning approaches.

avatar, and then the world is usually reset to a former state only to build a trap for the person who will build the trap."

"Exactly," the wizards continue, "The EXCALIBUR agent's planning system takes a different approach. It iteratively repairs *complete grounded plans* for which a concrete quality can be determined. The repair methods can exploit the quality information to improve the plan, e.g., by replacing a set of actions that cause a loss in quality. This technique, based on local-search methods, is not so – let's say *unfocused* – with respect to the plan quality, and there is practically no restriction on the kind of objective function to be optimized (see also [89]). The ASPEN system [31] is very similar to EXCALIBUR in this respect. To sum up, *decision-theoretic planning* is a very broad term, which, of course, also encompasses the EXCALIBUR agent's planning system. But the system is not what one would normally expect of a decision-theoretic planning system. So... well... you look a bit too drunk to go home now, don't you?"

This sounds like a good chance to have another jar of the ethanol potion with the wizards, and you gleefully agree. However, your impressive cleverness seems to have failed this time because you are not even quick enough to reply to the wizards' "Great, I'm always glad of opportunities to apply teleportation spells!" After a short puff, you find yourself in a trap you recently built yourself – together with a large number of angry and well-armed humans that were trapped while looking for an Orc who had just pestered some other humans.

1.4 Search Paradigms

In addition to the plan representation, a technique for building plans is needed. The task of plan construction can be formulated as a search problem, where we search in a space containing all possible plans for a consistent plan that satisfies our goal conditions[6]. There are two main paradigms for search: refinement search and local search. Figure 1.5 contrasts these paradigms.

1.4.1 Refinement Search

Refinement search is a stepwise narrowing process (see Fig. 1.5 (a)) alternating between **commitment** and **propagation**. In each refinement, a subset

[6] A common differentiation is made between a search in a state space and a search in a plan space. A search in a state space is based on a graph with world-state nodes that are connected by directed action arcs. The search tries to find a connection between a start node and a goal node. The resulting plan is the sequence of actions along the connection found. Within the usual framework of search techniques it is more common to use a strategy in which the search space's states are potential solutions (or plans) themselves. This is the approach of plan space search. Since a state space search can easily be expressed as plan space search, only plan spaces are considered here.

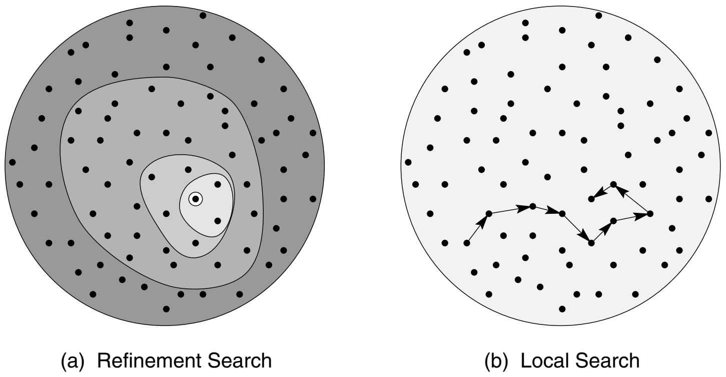


Fig. 1.5. Exploration of the Search Space

of states (or plans) is chosen for further investigation until a solution is found. The choice of a subset is made by a commitment to a special plan property. Many potentially chosen but unsatisfactory states can be excluded from further investigation as a consequence of the commitment's decisions. This cutback is performed by propagation methods.

Traditionally, refinement techniques apply a *complete* search. Because of the infinite number of possible states/plans, completeness can only be achieved if the choice of an infinite subset presumes investigation parallel to the rest of the set, provided the rest is not already under investigation or definitely does not contain solutions. If an investigated finite set lacks a solution, another set has to be chosen (usually by backtracking). Furthermore, the union of all potentially investigated finite sets must include all solutions.

For refinement planning, logic programming with its unification and backtracking provides a very expressive framework. Many approaches, such as the language $\mathcal{A}$ [63] or the language $\mathcal{E}$ [96], make use of it.

Typical instances of refinement planning are total-order, partial-order and hierarchical planning. Other approaches use maximal plan structures to attain an improved propagation behavior.

Total-Order Planning. Early planning systems constructed plans in a so-called total order. Starting with an empty plan, in each refinement step there is a commitment to a new action at a specific plan position. This position must be in a total order with respect to the plan's other actions. The propagation method deduces resulting intermediate states and excludes the next refinement step's choice options for actions that do not contribute to satisfying unsatisfied preconditions of the plan's actions or goals. For a new action, choice options for plan positions are excluded if the position is after the action/goal with the corresponding unsatisfied precondition or if there would be another action between the new action and the action/goal with the corresponding unsatisfied precondition such that the required effect would be nullified. The search terminates successfully if all preconditions of the plan's actions and goals are satisfied. Progressive planning with forward chaining is also possible.

Examples of total-order planners are STRIPS [55], WARPLAN [186] and Waldinger's planner [182].

Partial-Order Planning. Partial-order planning focuses on relaxing the temporal order of actions. In a refinement step, the position of a new action must not be totally ordered with respect to the plan's other actions. However, the commitment may include a decision on additional ordering relations that are necessary to ensure the consistency of the refinement. All unnecessary choice options for potential orderings are ruled out by the propagation process (sometimes called *least commitment*).

Partial-order planning is less committed than total-order planning, a total-order planning refinement subset always being a subset of (or equal to) a corresponding partial-order planning refinement subset. However, this does not necessarily result in the superiority of partial-order planning (see [120] for a detailed discussion).

Examples of partial-order planners are NOAH [161], TWEAK [28] and UCPOP [146].

Hierarchical Planning. The approach of hierarchical planning is to first introduce coarse-grained abstract actions which are then refined to the basic actions in a stepwise manner. All possible refinements are stored in a static transformation library.

Total- and partial-order planning apply a commitment to whole actions, including all of their pre- and postconditions. In hierarchical planning, the transformation library allows us to commit to only some pre- and postconditions (by way of abstract actions) and additional ordering information. Instead of introducing a new action, a refinement step consists of applying a transformation. The propagation allows only refinements according to the transformation library. Additional ordering relations must ensure the consistency of the refinement.

Hierarchical planning is highly dependent on an appropriate transformation library. The transformation library's hierarchical structure explicitly encodes inference information and guides search toward promising plans, but because of the limited library it is often far from being complete. An interesting aspect of hierarchical planning is its application of commitments other than those to whole actions.

Examples of hierarchical planners are ABSTRIPS [160], NONLIN [172] and SHOP [138].

Refinement Using Maximal Graphs. Total-order, partial-order and hierarchical planners create and reason about the plan's structures step by step. In contrast, planners like Graphplan [16] create a large maximal structure that includes all potential plans before starting the search process. Superfluous/inconsistent elements are then removed by the search process. The search can exploit these structures much better because propagation can involve the reasoning on parts of the plan for which no decisions have been made. However, maximal structures do not scale well. Other examples of this approach are *parc*PLAN [50] and CPlan [175].

1.4.2 Local Search

Local-search approaches perform a search by iteratively changing an initial state/plan (see Fig. 1.5 (b)). In each iteration, the **successor choice criterion** determines a state which will become the new search state. The potential successor states that can be chosen for a state are referred to as the state's **neighborhood**. The quality of the neighborhood states can be computed by an **objective/cost function**.

Neighborhood states are mostly states that differ in value switches of single variables or a simple extension/reduction of the plan. Of course, more complex changes are possible as well. For an optimization task, the neighborhood is usually constituted by feasible states only. For a satisfaction task, the neighborhood consists of partially consistent states.

Most of the local-search methods are incomplete, and it is possible for them to become trapped in local optima or on plateaus. Many local-search methods apply additional techniques to escape from these local minima and plateaus – such as tabu lists, random walks or restarts. With increasing dynamics of the agent's environment, the importance of these features decreases, as the search space changes quickly and less improvement is possible.

Local-search techniques include evolutionary algorithms [69], simulated annealing [102], tabu search [68], min-conflicts [121], GSAT [78, 165] and Walksat [166]. Applications to planning problems were proposed by Kautz and Selman [98, 99], Ambite and Knoblock [7], Muslea [125], Gerevini and Serina [67], Brafman and Hoos [23], and Chien et al. [31], among others.

1.4.3 Search-Paradigm Discussion

A key property of refinement search is that completeness can easily be realized. This is much harder to accomplish for local search, and mostly one cannot guarantee finding global optima or satisfying solutions. This also implies that proofs of global optima or inconsistency are beyond the reach of local search. Refinement search should therefore be the first choice for safety-critical problems.

Local search has other advantages. The production of a solution by refinement takes a lot of time, as a lot of labeling/propagation (and maybe backtracking) cycles have to be performed. In contrast, a local-search iteration can usually be computed very fast. This results in a temporally tight sequence of improvement steps, i.e., in an *anytime* behavior [195], and initial approximate results are computed much faster (see Fig. 1.6). Later on, when the search gets more constrained (less options lead to an increase in the objective/cost function's value) and the probability that local search gets caught in local minima increases, superiority is reversed. This is also confirmed by a couple of empirical studies, which report that short time limits and large problems usually mean that local-search methods are significantly superior to complete refinement methods (e.g., [183, 64]).

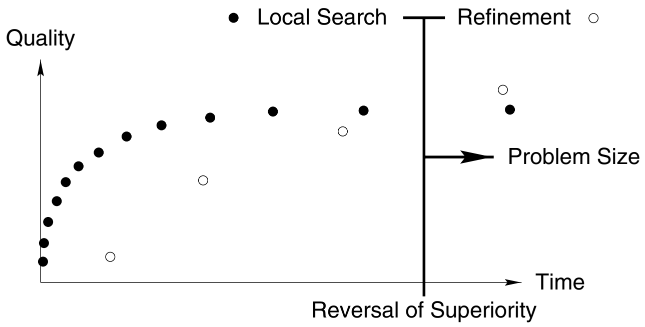


Fig. 1.6. Refinement Search vs. Local Search

A dynamic environment promotes local search as well because the search heuristics do not normally bother about modifications of the search space. This is different from complete methods, which have to update their memory of the already explored search space.

For a satisfaction task, local search's intermediate states are inconsistent. This means local search is inherently a partial constraint satisfaction [58]. An extension of refinement search with partial constraint satisfaction is much less efficient because the search space grows enormously.

To sum up, the application of local search should be considered if the problem domain features

– short computation time limits
– only restricted claims on optimality/satisfaction
– dynamics
– large problems[7]

This is definitely the case in a computer game environment, where the limited time available for AI computation (most of the CPU power is used for the game engine and graphics) and a complex and highly dynamic environment make reasoning properties like completeness and optimality only rough objectives to strive for. Furthermore, as argued in Sect. 1.3.2, local-search techniques can better exploit plan-quality information for realistic problems that have more complex objective functions.

[7] Of course, this list provides only general advice. The characteristics may vary depending on the specific search method and concrete problem.

1.5 Search Frameworks

As discussed in the previous section, local-search techniques for search and optimization are needed. There are numerous specific approaches for search, like neural networks or evolutionary algorithms. This section discusses *frameworks* for search, i.e., methods for specifying and formalizing search problems in a general manner. Specific solution methods can then be applied to solve these problems.

The advantage of using a general framework instead of specific approaches is the availability of ready-to-use off-the-shelf solution methods. In addition, future changes/extensions in the problem specification must be considered only at the modeling level and not in the underlying search algorithms. The main research areas that provide frameworks to formulate search problems are operations research, propositional satisfiability and constraint programming.

1.5.1 Operations Research

Operations research (OR) is a collection of techniques developed to handle industrial optimization problems. With our problem domain, we have mostly finite decision variables (there is no continuous transition between doing something or not). Continuous variables are only used for specific states (see also Chap. 4). The eligible operations-research methods for this domain are integer programming (IP) and mixed-integer programming (MIP) [139]. Problems are usually formulated using the following mathematical framework:

$$min\ \{c^{\mathcal{T}}x : Ax = b,\ x_1, \ldots, x_n \in \mathbb{N}\},$$

where c is the cost vector, x the vector of the decision variables $x_1, \ldots, x_n$, and $c^{\mathcal{T}}x$ the cost function that is to be minimized with respect to the equation system $Ax = b$. The decision variables have to be natural numbers, but in the case of MIP some of the variables can also be non-negative real numbers.

The solution methods are based on techniques like Simplex [34] and interior-point methods [97]. These approaches solve a linear relaxation of the problem (dropping the integrality constraints). Afterwards, the system is strengthened in order to enforce the integrality constraints, e.g., by a branch-and-bound with cutting planes.

There are only a few approaches for solving planning problems with OR methods. These include a domain-dependent approach by Bockmayr and Dimopoulos [18], ILP-PLAN by Kautz and Walser [101] and the state-change formulations by Vossen et al. [180].

1.5.2 Propositional Satisfiability

The problem of propositional satisfiability (SAT) is to decide whether a propositional formula is satisfiable or not. A propositional formula consists of two-valued variables (`true` and `false`) that are related via the operators $\neg$, $\vee$ and $\wedge$. Most solvers require that the problem be stated as a conjunctive normal form (a conjunction of disjunctions).

SAT-solving techniques include refinement approaches like the Davis-Putnam procedure [36] and Satz-Rand [75], as well as local-search methods like GSAT [78, 165], Walksat [166], HSAT [65], Novelty [116] and SDF [164]. Most of the SAT-based planning applications are based on the problem encodings of Kautz and Selman [99].

1.5.3 Constraint Programming

Unlike OR and SAT, constraint programming uses a much more declarative and general specification. Problems are formulated in a framework of variables, domains and constraints. The constraint satisfaction problem (CSP) consists of

– a set of **variables** $x = \{x_1, \ldots, x_n\}$
– where each variable is associated with a **domain** $d_1, ..., d_n$
– and a set of **constraints** $c = \{c_1, ..., c_m\}$ over these variables.

The domains can be symbols as well as numbers, continuous or discrete (e.g., “door”, “13”, “6.5”). Constraints are relations between variables (e.g., “x_a is a friend of x_b”, “$x_a < x_b \times x_c$”) that restrict the possible value assignments. Constraint satisfaction is the search for a variable assignment that satisfies the given constraints. Constraint optimization requires an additional function that assigns a quality value to a solution and tries to find a solution that maximizes this value. A brief example of constraint programming is given in Appendix B.

A satisfaction can be achieved by refinement as well as local-search approaches, already presented in Sect. 1.4 on search paradigms.

In the usual refinement approach, the variables are labeled (an assignment of a domain value) one after the other. In the case of an inconsistency (a constraint is violated), backtracking is triggered. Numerous variable- and value-ordering heuristics have been proposed, such as smallest-domain-first or smallest-value-first [162]. There are also generalized labeling techniques, in which the value selection is replaced by a domain reduction [73], and variable- and value-selection strategies with a stronger interleaving [94]. In addition, there are many extensions and modifications of naive backtracking [38, 150, 83], some of them being analyzed in [105].

The propagation of the refinement consequences is usually achieved by so-called consistency techniques. For example, we have two variables A and B with domains of $\{1 \ldots 10\}$ and a constraint $B > A$. In the case of a refinement of $A \in \{5 \ldots 10\}$, the propagation will entail a domain reduction of B to $\{6 \ldots 10\}$. The propagation can be made globally or locally. A global propagation analyzes all consequences within the constraint system and is very costly to compute. Local propagation is therefore usually applied, the consequences here being propagated only partially. Numerous local-propagation methods are available, such as arc-consistency algorithms like AC-4 [123] and AC-7 [15] or path-consistency algorithms like PC-4 [81].

Satisfaction mechanisms based on local search are not very popular so far. This may be due to the traditional logic-programming framework for constraint programming. Nevertheless, many new methods have been proposed, especially in the last years. This includes Hirayama and Toyoda's coalition forming [86] and the well-known min-conflicts heuristic for binary CSPs by Minton et al. [121] with its extension and generalization in GENET by Davenport et al. [35].

Applications of constraint programming to planning include Descartes [92], *parc*PLAN [50], CPlan [175], the approach of Rintanen and Jungholt [157] and GP-CSP [41].

1.5.4 Search Framework Discussion

The mathematical framework of operations research and the propositional formulas of SAT are highly restricted. The problems must be translated into a very specific form, which can only be done by experts. On the other hand, constraint programming allows the use of higher-level constraints, which make it easier to model a problem.

The OR and SAT approach is to break the problems down into their specific frameworks and to then scan the resulting specifications for structures on which to apply specific solution strategies. Although many efficient methods have been developed, the propositional clauses of SAT and the linear inequations of OR are scarcely able to exploit higher-level domain knowledge to support search (see also [119]). This is not the case for constraint programming. The advantage of this higher-level approach may be the reason for the superiority of constraint programming in domains like job-shop scheduling where powerful constraints are available – such as the `cumulative` constraint for resource-assignment problems that allows the application of specific technologies like edge finding [26, 27, 141, 12]. However, current constraint-based approaches that use local search do not normally exploit domain knowledge (see Chap. 2).

The SAT approach does not have numbers in its representation repertoire. The propositional SAT variables can therefore represent only qualitative differences – unlike CP and OR which can also represent quantitative information. Of course, discrete numbers can also be modeled in SAT by using a propositional variable for each value of the discrete domain. But this is a highly unsuitable approach, not only because of its costs but also because the values' ordering relation is often exploited during search.

Another reason for not using OR is that there are lots of planning decisions with discrete alternatives, and the performance of OR methods declines sharply as the number of integer variables increases. There are initial approaches aiming to combine constraint programming and OR solution methods (e.g., [17, 88]), but because the combination of these paradigms is not our main concern, we confine our contention to constraint programming. Some tentative work has also been done on combining SAT with linear inequalities (see [191]).

1.6 Conclusion

This section recapitulates on the main design decisions of the previous sections and draws conclusions for further development needs.

The agents must be able to exhibit sophisticated behavior and find solutions even for unforeseen situations. A *planning system* is therefore needed to guide an agent's behavior.

The system will be fully based on the *constraint programming* paradigm. This allows the agent's behavior problem to be represented by a declarative high-level specification that can be easily maintained. The application of high-level constraints allows domain-specific knowledge to be exploited for the search process.

Many planning systems already incorporate some kind of constraint handling. They can be grouped into the following categories:

– Planning with Constraint Posting:
  During a conventional planning process, constraints can be posted. Constraint satisfaction is used here as an add-on to check the satisfaction of restrictions such as numerical relations. For example, MACBeth [72] and the planning procedure of Jacopin and Penon [91] apply this scheme. Another approach is implemented in the Descartes system [92], which uses constraint postings not only for numerical values, but also for postponing some of the decisions of action choice.
– Maximal Graphs:
  A restricted planning problem is encoded as (so-called *conditional*) CSP. The CSP is constructed such that it includes all possible plans up to a certain number of actions, which can be activated or deactivated. *parc*PLAN [50], CPlan [175], the approach of Rintanen and Jungholt [157] and GP-CSP [41] follow this line. The advantage of this procedure is that decisions can easily be propagated throughout the CSP. However, the optimal size (e.g., the number of actions) is not known in advance, so a stepwise expansion of the CSP structures must be performed if no solution can be found. However, maximal structures scale *very* badly and are only feasible for small and only slightly variable structures, which is not the case with most planning problems.

None of these approaches makes it possible to represent the planning completely within constraint programming. The problem here is the restrictiveness of conventional formulations for constraint satisfaction problems because all the elements and their relations, i.e., the *constraint graph*, have to be specified in advance. But plans are highly variable and it is not possible to predict which actions will be used in which combination. The partially observable computer-game environment of an agent poses a further problem here because there are an unknown number of objects. An extension of CSP formulations must therefore be developed that can handle variable constraint graphs. This is done in Chap. 3.

The next problem faced, if constraint programming is to be applied to computer games, is that only a very small portion of CPU power is left for the necessary computations – and there is more than one agent to be guided. Computation here is not a matter of hours, days or seconds but rather clock-ticks. This is one reason for applying *local-search* techniques in an agent's planning system. The iterative optimization of local search enables the agent's anytime reaction and fast adaptation to changes in the environment. The more computation time there is available, the more elaborate the agent's plan will get.

The current approaches that combine local search with constraint satisfaction use only very basic constraint types, corresponding rather to SAT- or OR-based local-search approaches. The drawback of these approaches is that they fail to take a global view of the problem and are unable to provide appropriate search guidance. The use of higher-level constraints for local search is needed, which is also more in line with the fundamental principles of constraint programming. Chapter 2 addresses this problem.

Besides the extensions of the constraint programming framework mentioned above, the use of constraint programming for planning requires that appropriate high-level constraints be available to model planning problems, which must include the handling of temporal and other resources. Chapter 4 describes the basic constraint types for specifying planning problems and their use. In addition, issues of probabilities and incomplete knowledge are treated on the modeling level.

2. Using Global Constraints for Local Search

Our agents need to adapt their behavior in real time to new situations. Conventional refinement-based constraint programming techniques are not suitable for these requirements. Thus, as discussed in Sect. 1.4.3, our agents will be based on local search. This chapter presents a technique to combine local search with constraint programming.

The use of local search has become very popular in recent years as applications have begun to tackle complex real-world optimization problems for which complete (refinement) search methods are still not powerful enough.

Conventional ways of using local search are difficult to generalize. Increased efficiency is the only goal, generality often being disregarded. It is quite common to define highly sophisticated and problem-tailored representations with specialized neighborhoods and successor selection methods (see [1] for examples). From a software engineering point of view, this is not a good idea. Integrating complex, heterogeneous problems in a monolithic system hinders the system's reuse, extension and modification.

Other approaches take the general constraint programming framework as starting point and try to introduce local search methods for constraint satisfaction. Through the use of CSP formulations, local search acquires a general application-independent framework. CSP formulations used with local search approaches typically involve only very basic constraint types, e.g., linear inequalities or binary constraints. The problem with this kind of formulation is that the inherent problem structure is mostly lost by the necessary translation of the problem to this low-level formulation. Domain-specific knowledge about appropriate representations and search control is only available at a higher level and cannot be used. Consequently, these methods frequently fail because they have only a very limited view of the unknown search-space structure.

This work attempts to overcome the drawbacks of these two local search approaches – monolithic problem-tailored and general low-level – by using global constraints. The use of global constraints for local search allows us to revise a current state on a more global level with domain-specific knowledge, while preserving features like reusability and maintenance. The proposed strategy is demonstrated on a dynamic job-shop scheduling problem, which is a subproblem of an agent's planning task.

Local search techniques provide a solution at any time, the quality of the solution being subject to constant improvement. This anytime feature makes it worthwhile evaluating the whole solution process, not just the final result. To have a measure for the utility of intermediate states as well, we can extend the CSP formulation to weighted CSPs (WCSPs). In WCSPs, constraints have associated costs, which are dependent on the constraint variables' assignments. The goal is to minimize the total sum of costs[1], which provides us with a useful cost function for local search. A value of zero for the costs means total satisfaction. This graded consistency measure is essential for our approach, as we cannot expect to achieve an agent's total satisfaction because of the limited reasoning time.

2.1 Global Constraints

Constraint satisfaction has traditionally been tackled by refinement search (domain-reduction techniques), which faced similar problems of exploiting high-level knowledge. The success of commercial tools like the ILOG Scheduler [111] can be ascribed mainly to their use of so-called *global constraints*. Régin [155] defines a global constraint as a substitute for a set of lower-level constraints, such that a more powerful domain-reduction algorithm can be applied. Using global constraints, constraint solvers can attain an enormous speedup and real-world application requirements can be satisfied.

For example, the `alldifferent(V)` constraint of the "send more money" example in Appendix B is a global constraint that forces all included variables of the set `V` to take different values. A formulation by lower-level constraints would include inequality constraints for all possible variable pairs of the set `V`. But the application of the global constraint with a specific data representation and satisfaction methods can yield much better performance (e.g., by a demon-observed array representation [152]).

The notion of global constraints can support local search approaches as well. It is transferred to a local search context in the following subsections.

2.1.1 Global Constraints from a Local Search Perspective

The central issue in local search is the transition from one state to the successor state. As it is uncertain what kind of change improves a current state, lots of neighbor states are usually analyzed. There are no general rules here, the kind of neighborhood and successor selection being a heuristic matter.

The term *heuristic* already implies the existence of some *domain knowledge* for guidance. The heuristics of conventional local search mechanisms for CSPs, like GSAT [78, 165], can only exploit knowledge about their representation's special low-level structure, thus having to cope with self-produced

[1] The term *inconsistency* is used below as a synonym for *costs*.

complications instead of being able to incorporate higher-level domain knowledge. It is well known that there is a strong relation between a problem's representation and its computability, and it remains unclear to what kind of problems these low-level standardized representation approaches are suited. Consequently, there is often not enough information locally available to direct the search [121]. Larger variable domains than the binary ones normally used exacerbate the problem because information about qualitative differences is not directly available.

The concept of global constraints can help remedy this situation. To this end, we have to extend the notion of a global constraint:

> A global constraint is a substitute for a set of lower-level constraints, additional domain knowledge allowing the application of specialized data representations and algorithms to guide and accelerate search.

A global constraint must feature a start function to construct its initial structures and inconsistency, an improvement procedure for determining a promising successor state, and an update function for its internal structures and the constraint's cost contribution in case of variable changes.

Heuristics for building neighbors and for selecting a successor can be directly encapsulated in the global constraints, where the appropriate domain knowledge is available. This is illustrated in the following sections using an example of a global constraint PERMUTATION. The global constraint PERMUTATION(A, X) has to ensure that the variables of set X are a permutation of the values of bag A. The costs of the constraint are to be determined by the minimal sum of the distances of the variables' current values to a valid assignment.

2.1.2 Internal Structures

Many successful applications of local search gain their power from efficient updates of the inconsistency information rather than from recomputations from scratch. For this purpose, *additional structures* often have to be maintained. In the case of the PERMUTATION constraint, three lists can be used as additional support structures: list l_a with the ordered values of bag A, list l_x with the variables of X ordered by their current assignments, and list l_c with the i_{th} element being the distance between the i_{th} value of l_a and the value of the i_{th} variable of l_x. The costs of the PERMUTATION constraint can be computed by summing the values of l_c.

For PERMUTATION(<1, 7, 4, 3>, {A, B, C, D}) and a current assignment of A=2, B=6, C=2 and D=7, the structures are constructed in the following way:

$$\begin{aligned} l_a &= [1,3,4,7] \\ l_x &= [A_{(2)}, C_{(2)}, B_{(6)}, D_{(7)}] \end{aligned}$$

$$\begin{aligned} l_c &= [|1-2|=1, |3-2|=1, |4-6|=2, |7-7|=0] \\ Permutation_{costs} &= 1+1+2+0=4 \end{aligned}$$

Note that by changing a variable all global constraints involving this variable must be called to update their internal structures. For the PERMUTATION constraint, there must be a reordering of the variable in l_x with a corresponding update of l_c and the resulting cost sum. For example, if B changes from 6 to 9,

$$\begin{aligned} l_a &= [1,3,4,7] \\ l_x &= [A_{(2)}, C_{(2)}, \underline{D_{(7)}}, \underline{B_{(9)}}] \\ l_c &= [1,1,\underline{|4-7|=3}, \underline{|7-9|=2}] \\ Permutation_{costs} &= \underline{4+(-2+3)+(-0+2) \ = \ 7} \end{aligned}$$

2.1.3 Improvement Heuristics

What is a good heuristic for building the successor state may depend on the other constraints involved as well as on the current state of the search. Thus, a global constraint should provide various heuristics, e.g., one with a complete revision of the violated variables, one with a minimal change of just one variable, one with a randomized successor selection, and one with random walks.

For the PERMUTATION constraint, the list l_c gives us quantified information about the variables' violation of the constraint. A heuristic with a cautious strategy can reset only one variable – according to the highest element in l_c – to the corresponding value of l_a. Following the last assignment of A=2, B=9, C=2 and D=7, the variable D would be changed to 4:

$$\begin{aligned} l_c &= [1,1,\underline{3},2] \\ l_a &= [1,3,\underline{4},7] \\ l_x &= [A_{(2)}, C_{(2)}, \underline{D_{(4)}}, B_{(9)}] \\ Permutation_{costs} &= \underline{7+(-3+0) \ = \ 4} \end{aligned}$$

The range of heuristics to be applied to a specific problem should be customizable by the user. During search, a constraint-internal function has to decide on the concrete heuristic to be applied. This decision can be taken at random, be dependent on the current search state, or be subject to learning mechanisms.

2.2 Granularity

If the PERMUTATION constraint is to be modeled by low-level constraints, linear inequalities can be used. A particular problem, e.g., PERMUTATION(<1, 7, 4, 3>, {A, B, C, D}), can be expressed by the following CSP:

$$A, B, C, D \in \{1, 3, 4, 7\}$$
$$\forall n \in \{1, 3, 4, 7\} : A_n, B_n, C_n, D_n \in \{0, 1\}$$

$$\begin{aligned} A &= A_1 + 3 \times A_3 + 4 \times A_4 + 7 \times A_7 \\ C &= C_1 + 3 \times C_3 + 4 \times C_4 + 7 \times C_7 \\ B &= B_1 + 3 \times B_3 + 4 \times B_4 + 7 \times B_7 \\ D &= D_1 + 3 \times D_3 + 4 \times D_4 + 7 \times D_7 \end{aligned}$$

$$\begin{aligned} A_1 + A_3 + A_4 + A_7 &= 1 \\ B_1 + B_3 + B_4 + B_7 &= 1 \\ C_1 + C_3 + C_4 + C_7 &= 1 \\ D_1 + D_3 + D_4 + D_7 &= 1 \end{aligned}$$

$$\begin{aligned} A_1 + B_1 + C_1 + D_1 &= 1 \\ A_3 + B_3 + C_3 + D_3 &= 1 \\ A_4 + B_4 + C_4 + D_4 &= 1 \\ A_7 + B_7 + C_7 + D_7 &= 1 \end{aligned}$$

The permutation problem is very hard to recognize now. And not even the global constraint's distance measure is included in the upper solution. In contrast to the low-level formulation, the statement of a global PERMUTATION constraint is highly declarative and easy to understand.

A local search method that is based on a low-level problem representation cannot make use of domain knowledge and is not able to recognize the low-level constraints' interactions, e.g., that it is inadvisable to compensate a change from $A_7 = 1$ to $A_7 = 0$ with respect to the constraint $A = ...$ by an activation of $A_3 = 1$ and $A_4 = 1$ in order to keep the constraint satisfied. The lack of a general overview and of heuristic knowledge makes the low-level approach less efficient and susceptible to cycling and getting stuck in local minima.

A low-level representation also has advantages, though. A wide variety of problems can be modeled using the general low-level representation, whereas modeling using global constraints presupposes the availability of suitable global constraints. For example, if the problem is to find an assignment such that the variables of set X are a permutation *of a subset* of the values of bag A, the low-level representation can easily be adapted. But a solution based on the global PERMUTATION constraint would require a considerable effort to adapt the constraint's internal structures and heuristics, if not a redesign from scratch[2].

A comparison of global constraints and monolithic solutions is straightforward, regarding the global constraints as a low-level representation and the

[2] However, all low-level constraints could be realized by global constraints as well, enabling the user to model the problem by these constraints if no high-level constraint is available

monolithic solutions as a single highly global constraint. Again, the higher-level (monolithic) system is faster as it makes better improvement decisions because of its superior overview, but it is difficult to reuse because of its specialization.

Global constraints are a compromise between the generality of low-level CSP-based local search and the efficiency of monolithic problem-tailored local search encodings (see Fig. 2.1). Finding the *right* granularity for global constraints is up to the designer. Only very general rules apply, which are comparable to the problems of object-oriented design (see Gamma et al. [60] for a discussion and general guidelines).

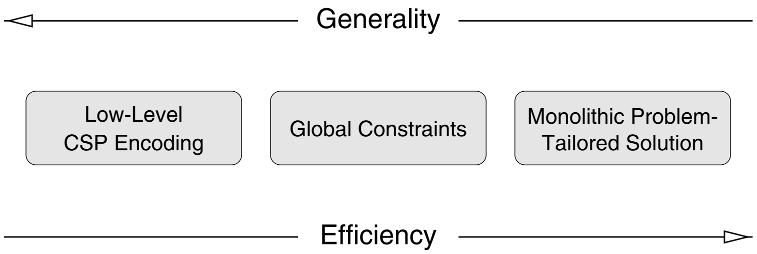


Fig. 2.1. Global Constraints as a Compromise

2.3 Global Search Control

As described above, global constraints have integrated heuristics to enable them to choose a successor state on their own. On top of the constraints, there must be a mechanism that combines the constraints' cost contributions to the overall cost function and a regulation determining which constraint is allowed to select the successor state for a current iteration. This is the job of the *global search control*. The components' interplay is outlined in Fig. 2.2.

The global search control serves as a general manager, to which variables and constraints can dock on and off in a plug-and-play manner. This provides a simple mechanism for tackling problems with dynamic changes. To add a new variable, the variable in question must already be instantiated. To add a constraint, the constraint's inclusion of a variable must be announced to the variable to allow the constraint's updating in the case of value changes.

For the overall cost function, which is handled by the global search control, problem-specific coefficients can be introduced to weight the constraints' subjective costs, as a constraint's importance may be different in different contexts (see also Sect. 2.4.3).

Conventional local search methods make a successor decision on the basis of a calculation of the neighborhood states' costs. These calculations might

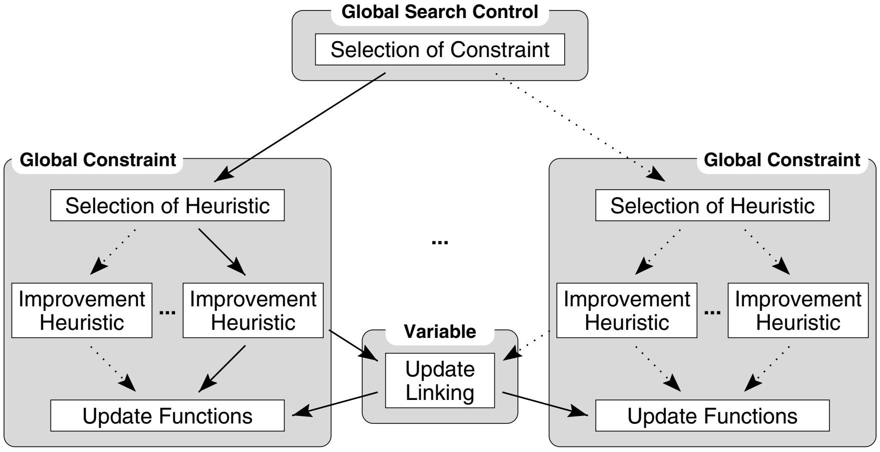


Fig. 2.2. Local Search with Global Constraints

be quite costly to compute within the global constraint framework, but the current constraints' costs can serve as an excellent compensation for this. As the goal is to minimize the constraints' costs and as the global constraints should know how to resolve the conflicts by themselves, it is a straightforward matter to select constraints according to their current inconsistency.

The selection of the global constraint that is to improve the current state can be enhanced by various meta-heuristics to avoid local minima and plateaus, ranging from a simple random choice of unsatisfied constraints to more elaborate techniques including learning, tabu lists, etc. Some of these are demonstrated in the following section's case study. The global search control module should support a variety of methods that can be applied in a user-specified way (as in the flexible blackbox system [100]). By integrating global search-state parameters for the constraints' improvement procedures, e.g., a simulated-annealing-like temperature, an anytime improvement depending on the current search situation can be achieved. In addition, it is possible to introduce higher-level coordination mechanisms between constraints to minimize violations of multi-constraint variables. If knowledge is available about these interface areas, it may be sufficient to introduce redundant global constraints to cover these areas. Nevertheless, the overhead of coordination mechanisms may pay off in some cases.

2.4 A Case Study: Dynamic Job-Shop Scheduling

As an example of the application of local search with global constraints, we look at the dynamic job-shop scheduling problem.

The problem of solving standard job-shop scheduling by local search has been addressed in numerous research papers (see [174] for a survey), but there are few experiments dealing with a partial satisfaction/infeasible state neighborhood. Experiments including dynamic aspects of job addition/removal are also rare. This is a pity since dynamics and partial satisfaction are very realistic problem features and local search approaches can bring their advantages to bear in these domains particularly.

As in traditional $n \times m$ job-shop scheduling, there are n jobs in a schedule, each of them having m tasks with fixed durations. All tasks of a job are connected via $m - 1$ linear distance inequalities involving start or end time points of two tasks. For compliance with the scheduling horizon h, there is in addition a linear distance inequality $task_{end} \leq h$ for each task. Each task has to be processed on a specific machine, and each of the o machines can process only one task at a time. Every p microseconds of computation time, one job is removed from the schedule and a new job must be added. The tasks of the new job are randomly distributed within the scheduling horizon.

The goal is to find a concrete begin/end for all currently active tasks, such that the inconsistency of the constraints is as low as possible. To compare different algorithms, the inconsistency can be displayed over time (see Fig. 2.3; peaks occur on job removal/addition) and the average inconsistency can be computed.

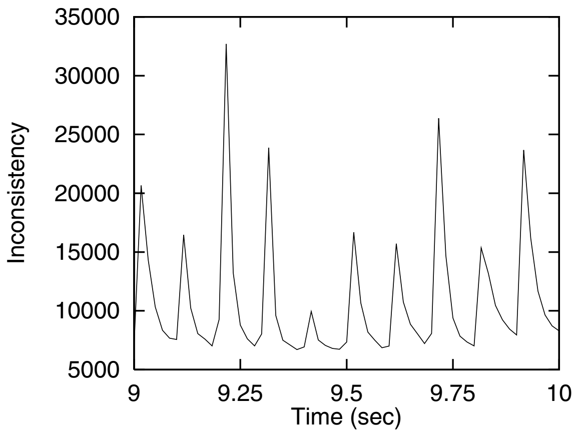


Fig. 2.3. A Test Run Example

The measurement of inconsistency is done in the following way:

– For each discrete point of time from 0 to h on each machine, the *number of assigned tasks* -1 is added to the inconsistency if there is an assignment of more than one task.
– For each linear distance inequality that is unsatisfied, the *minimal shift distance* for one of the inequality variables required to satisfy the inequality is added to the inconsistency.

The initial state for the test runs is computed by the iterative addition of n jobs, with p microseconds of computation time between each addition, which is used for improvement iterations.

The following parameters were used for the test runs throughout our experiments: The tasks' duration is 100 plus a random value of between -99 and +100. 40 % of a job's tasks require the same machine as another of the job's tasks. The jobs' inequalities consist of 20 % equations and 80 % real inequalities. The inequalities involve a shift constant, which is 100 plus a random value of between -99 and +100.

2.4.1 Realization

The dynamic job-shop scheduling problem was encoded according to the global constraint concept[3]. There are two types of global constraints:

- Action Resource Constraints (ARCs) for the nonoverlap of tasks that require the same machine
- Task Constraints (TCs) for the temporal ordering relations between tasks within a job

Decomposition into these types of constraints is done because of the strong dependencies among the variables for each constraint and the manifold reuse possibilities. Resource constraints that hinder a temporal overlap of activities/assignments are needed in many applications, e.g., to model the availability of a person, a room or a machine, and the TC's temporal relations of tasks/activities are needed in nearly every system that involves temporal reasoning. The ARCs and TCs will also be used for the agent's planning system.

Global Action Resource Constraints. An ARC is connected to a set of pairs P and two variables c and h. A pair of the set P, (s_i, e_i), represents a task i that uses the machine/resource. The starting time of a task i is represented by the variable s_i and the end of the task by the variable e_i. The ARC's role is to prevent overlapping tasks. For each discrete point of time from c to h, the *number of assigned tasks* -1 is added to the constraint's inconsistency if there is an assignment of more than one task at this time point. The constraint can change the tasks' temporal distribution by changing the variables s_i/e_i.

An ARC's basic internal structures are a set of task objects and a multiple-linked list of temporal intervals. A task object contains references to the start variable, the end variable and its intervals. The intervals split the time from c to h into maximal parts such that each interval has the same task assignment for its duration (see Fig. 2.4). Task overlaps are mapped to the intervals, and the inconsistency of an interval is computed by the overlaps multiplied by the interval's length. Links to an interval's task objects are also stored. The ARC's total inconsistency is the sum of the intervals' inconsistencies.

[3] The implementation was done in ObjectC.

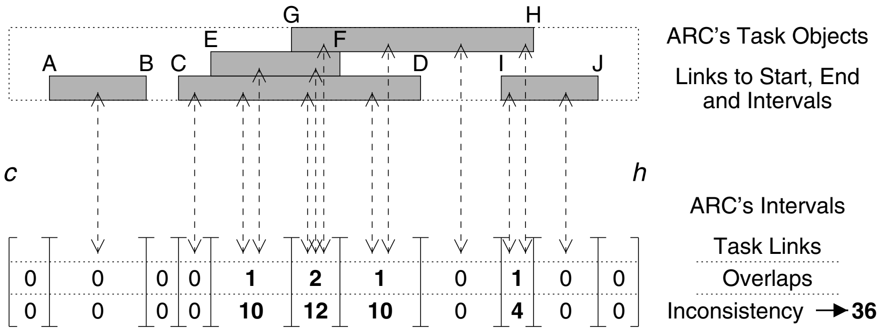


Fig. 2.4. An ARC's Internal Structures

The ARC's basic heuristic (ARC-H1) selects an inconsistent interval (i_1; see selection a) of Fig. 2.5) with a choice probability for an interval that is proportional to the interval's inconsistency. For example, if the intervals A to E have inconsistencies of $A_{costs} = 10$, $B_{costs} = 12$, $C_{costs} = 10$, $D_{costs} = 0$ and $E_{costs} = 4$, the chance of being chosen is 27.8 % for A, 33.3 % for B, 27.8 % for C, 0 % for D and 11.1 % for E.

Then, one of the interval's tasks (t_1) is chosen at random. The task's start variable is shifted to the beginning of an interval i_2, a choice probability for the interval being proportional to its length times its task-number improvement with respect to the interval i_1. Only intervals with fewer tasks than the i_1 interval's (without the task t_1) are considered for this decision, and the maximal length of intervals considered for the multiplication is that of the task t_1.

Global Task Constraints. A task constraint manages a set of temporal relations between a job's tasks (see Fig. 2.6). Each relation involves links to two decision variables V_1 and V_2, a constant c and a comparator $\bowtie \in \{<, =, >\}$, such that $V_1 \bowtie V_2 + c$. The inconsistency of a relation is given by the minimal shift distance for one of the variables required to satisfy the relation. The TC's total inconsistency is the sum of the relations' inconsistencies.

The basic improvement heuristic of the task constraint (TC-H1) selects an inconsistent relation with a choice probability for a relation that is proportional to its inconsistency (see Fig. 2.7). One of the involved variables is selected randomly, and a minimal shift of this variable is performed such that the relation is fulfilled.

2.4.2 Results

Figure 2.8 shows experiments with different horizons. The global search control selects a constraint with a probability proportional to the constraint's costs. The schedule always contains 50 jobs (→ 50 TCs), each of them with

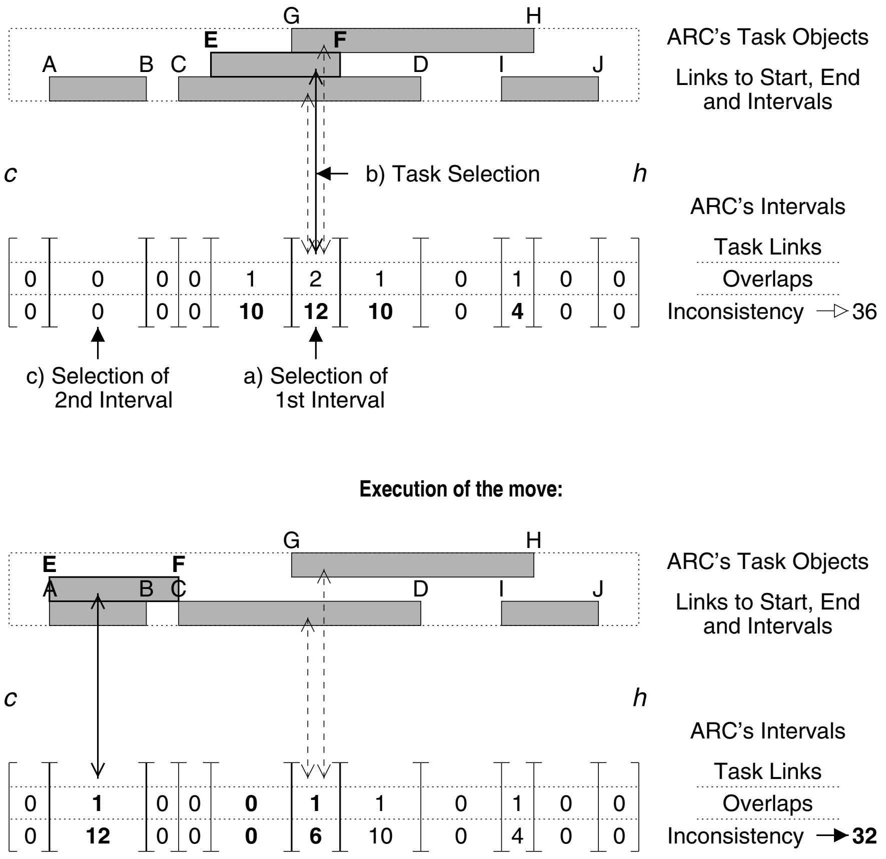


Fig. 2.5. The ARC-H1 Heuristic

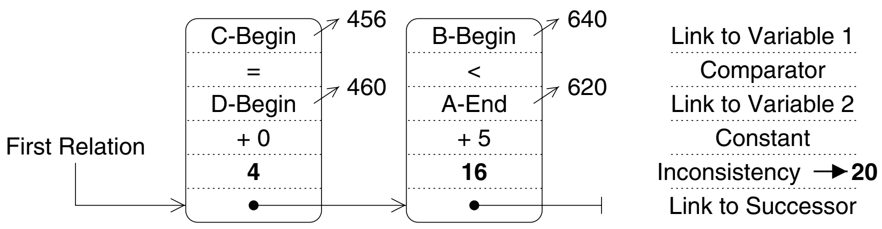


Fig. 2.6. A TC's Internal Structures

five tasks, and there are five machines (→ 5 ARCs). Every tenth of a second there is a job removal/addition[4].

[4] Although there is an enormous variety of possible problem configurations, the results are presented for just one problem instance (given as the average of 1,000 test runs if not otherwise stated). Comparable results were obtained with other instances.

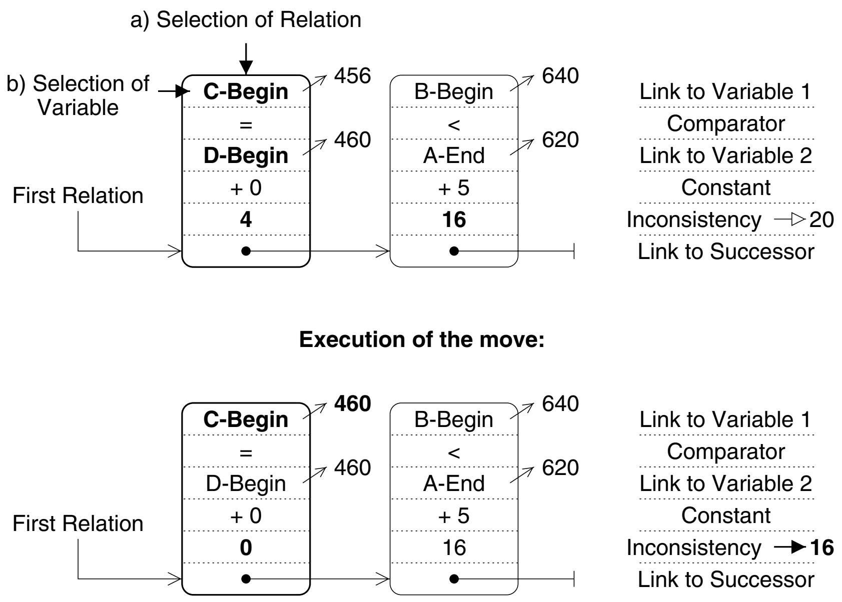


Fig. 2.7. The TC-H1 Heuristic

According to a computation using refinement/global search by the ConPlan system [74], the minimal horizon for a maximal consistent solution varies around 6,000, depending on the currently active jobs[5]. With a horizon of 2,000, the topology of the search space is so flat that any improvement effect is close to pure noise. A more complete picture is given in Fig. 2.9. One point represents the inconsistency averaged over 10 seconds of runtime.

In order to study the search behavior until complete satisfaction was achieved, no job removal/addition was done for the rest of the experiments. There are 10 jobs ($\rightarrow$ 10 TCs) with 10 tasks, 10 machines ($\rightarrow$ 10 ARCs) and a horizon of 2,300. The start inconsistency is about 50,000.

2.4.3 Constraint Weights

The weights of the constraints' subjective costs for the overall cost function have so far been set to one. Looking at the individual constraints' cost development for a test run (Figure 2.10) may give the impression that this is not the best choice. Better weightings could restructure the search space and ensure that the inconsistency of possibly more critical constraints plays a more important role and that these constraints are chosen more often to execute improvement changes.

[5] A performance comparison with the ConPlan system involving the dynamics was not possible as a tenth of a second was not even enough to set up the constraints.

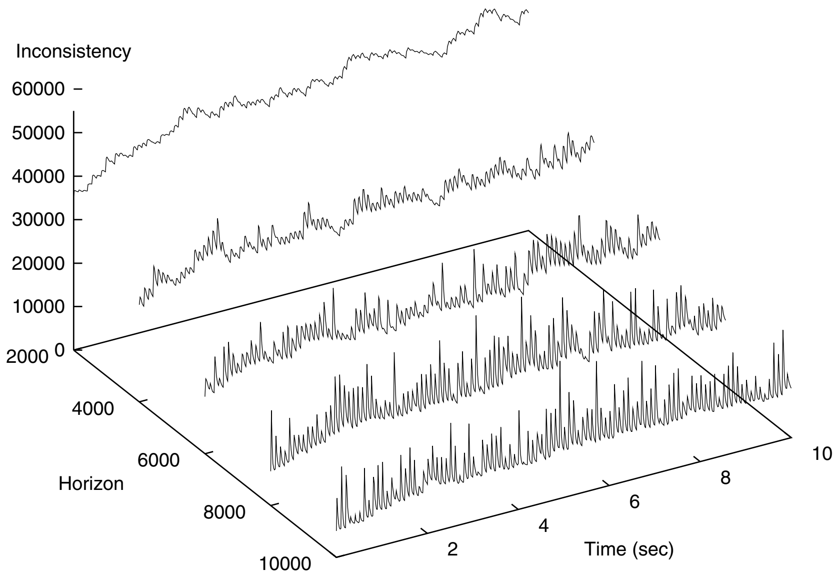


Fig. 2.8. Horizon Variation – Single Runs

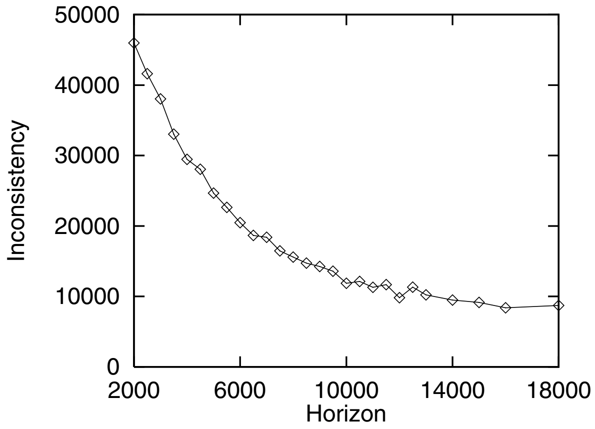


Fig. 2.9. Horizon Variation – Averages

Figure 2.11 shows that this is not the case. The weights for the ARC and TC constraints' costs are given after the colons[6]. A stronger weighting of

[6] To be able to better compare the differently weighted inconsistencies, the results are presented by displaying the inconsistencies with constraint weights of one. As

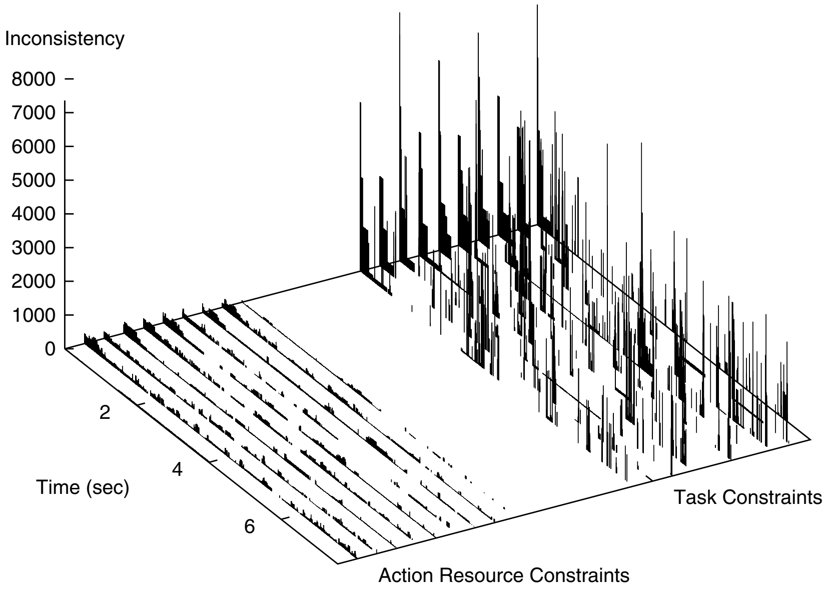


Fig. 2.10. Cost Distribution for a Single Test Run

the ARCs has a negative effect, whereas higher weights for the TCs neither worsen nor improve the search behavior.

The ratio of iterations with task constraint selections to all iterations is shown in Fig. 2.12. *NgRmRt*, *NgNmNt*, *RgNmNt* and *RgRmRt* are test-run variants in approximately decreasing order of quality (see Sect. 2.5.1). The high initial rates are due to the great disorder of the randomly distributed tasks. The large variation toward the end is due to the fact that there is less data involved, most of the test runs having already finished. The higher the quality of the test runs, the higher too the ratio of iterations with task constraint selections. This underpins the assumption that a higher repair rate of the ARCs does not improve the search behavior.

the line patterns are sometimes difficult to recognize, the legends list the curves in the lines' order from top to bottom.

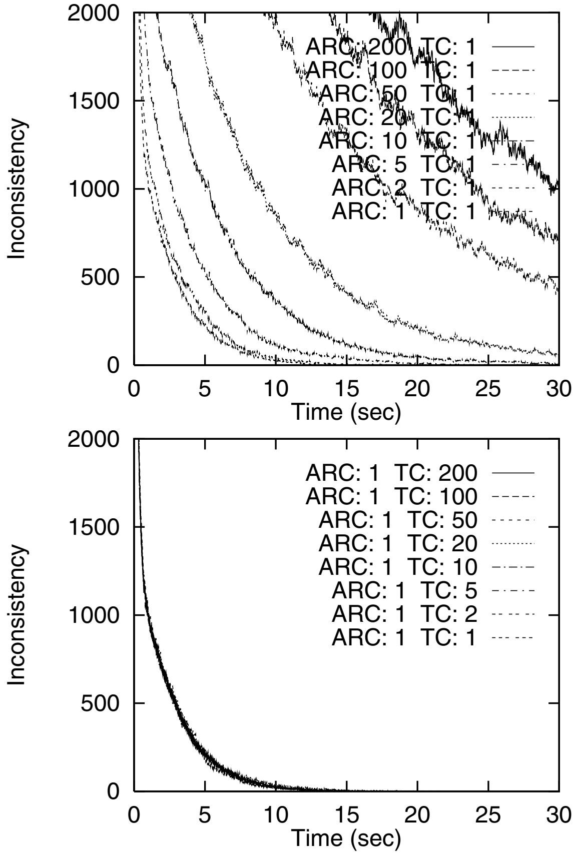


Fig. 2.11. Variation of Weights

2.5 Susceptibility to Local Minima and Plateaus

Unlike low-level CSP-based representations, global constraints enable the search to be conducted in a more informed way. A measure for this is the susceptibility to getting caught in local minima and on plateaus. This is investigated in the following subsections.

2.5.1 Randomization

The previous test runs had randomization at all choice options. This is a common technique to leave local minima and plateaus. However, randomization need not always have a positive effect. Figure 2.13 shows choice variants, where N means choosing the subject with the highest inconsistency, and R means a choice with a probability of a subject's being chosen that is pro-

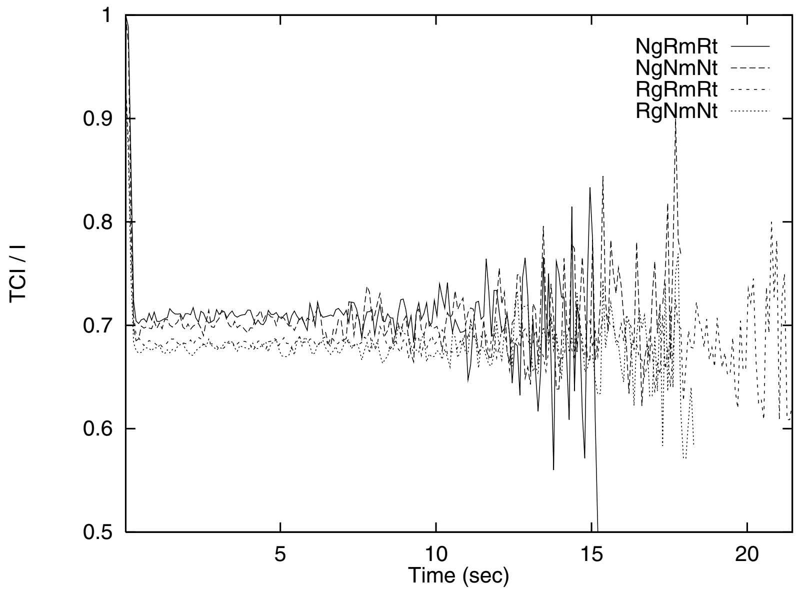


Fig. 2.12. Ratio of Task Constraint Selections

portional to the subject's inconsistency[7]. The letters g, m and t indicate the choice points: g the global search control's constraint selection, m the first interval choice of the ARCs' improvement heuristic (see Fig. 2.5), and t the relation choice of the TCs' improvement heuristic (see Fig. 2.7).

For the first phase (Figure 2.13, top graph) of the search, the quality of the strategies can be more or less ordered according to their amount of randomization, the nonrandomized *NgNmNt* version clearly being the best. The superiority of nonrandomized strategies is not surprising, as local minima and plateaus are less probable in the early phase. The *Ng* component has the most important impact. Strategies with an *Rg* component are nearly always worse, even in the later phases of the search (middle and bottom graphs).

As the search proceeds, the *Rm* component becomes more important, indicating that the ARC's heuristic no longer always makes the right decisions. Shortly after, the *Rt* component acquires some influence as well, making *NgRmRt* the first to converge to a complete satisfaction, followed by the nonrandomized *NgNmNt*. *RgNmNt* is the third to achieve complete satisfaction, only a little before *NgRmNt*.

In general, nonrandomization seems to be best for the g decision, whereas randomization of m and t depends on the available computation time. The randomization of m is much more important than that of t, which indicates that the ARC's heuristic is not very powerful.

[7] See Appendix C for a discussion of techniques for randomized choices.

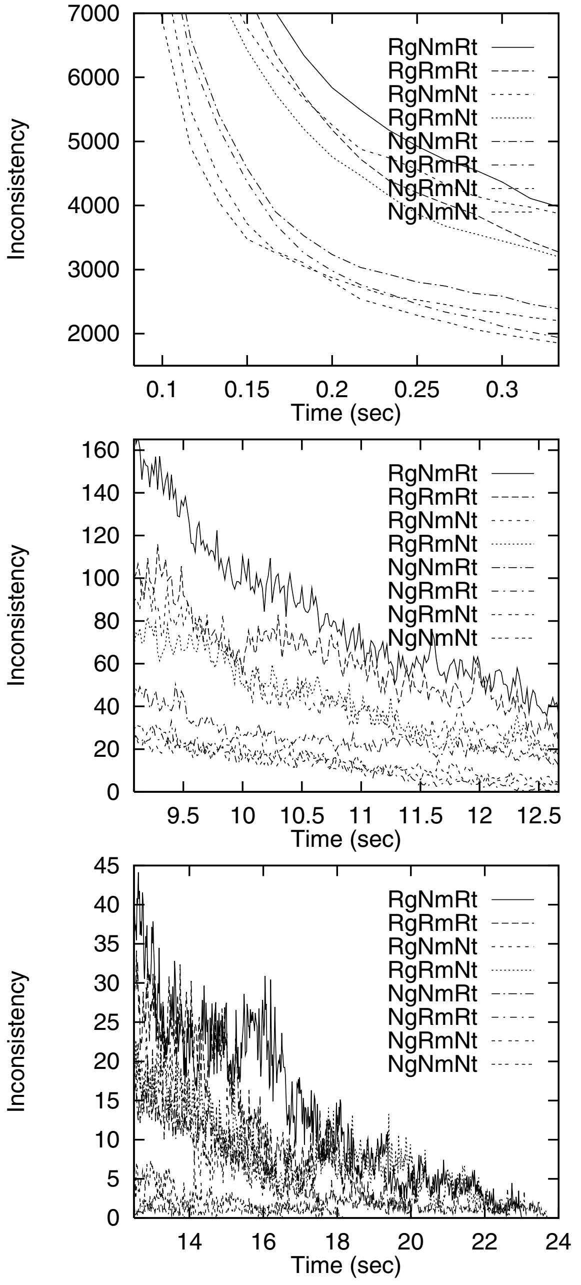


Fig. 2.13. Randomization Variants

One should be careful with anytime switching between variants for different search phases based on the graphs of the individual variants. The switch to a variant with the steepest descent for an actual inconsistency does not necessarily represent the optimal behavior, because each variant has a different search history and may require structurally very different areas of the search space in order to advance. A prognosis of the behavior of switching strategies is further complicated by the dynamics of the dynamic job-shop scheduling problem.

2.5.2 Random Walks

Random walks are random moves in the search space that disregard the change of the cost function value. The idea is that the search is retracted from hopeless situations (like local minima) from time to time. Unlike restarts, random walks remain within the area of the current state.

Random walks can be included by introducing a second improvement heuristic for each constraint that makes a random variation of a random variable. Figure 2.14 shows the results for different probabilities for the random variation heuristics to be chosen. It is obvious that the random walks generally cause a deterioration in the results. At no point is there a cross-over: the more random walks, the more inconsistency.

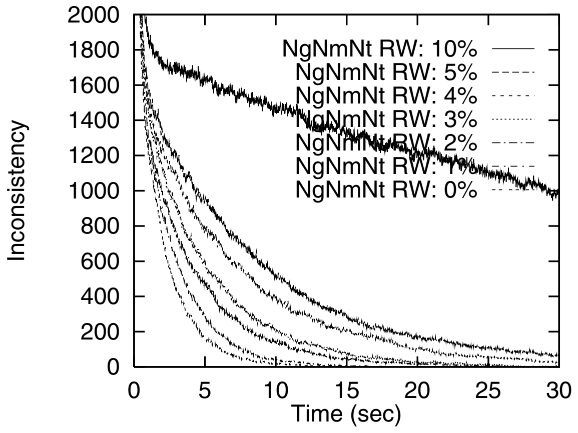


Fig. 2.14. Random Walks

2.5.3 Tabu Lists

Figure 2.15 shows experiments using a tabu list for the global search control's constraint selection (based on tabu search [68]). Each selected constraint is

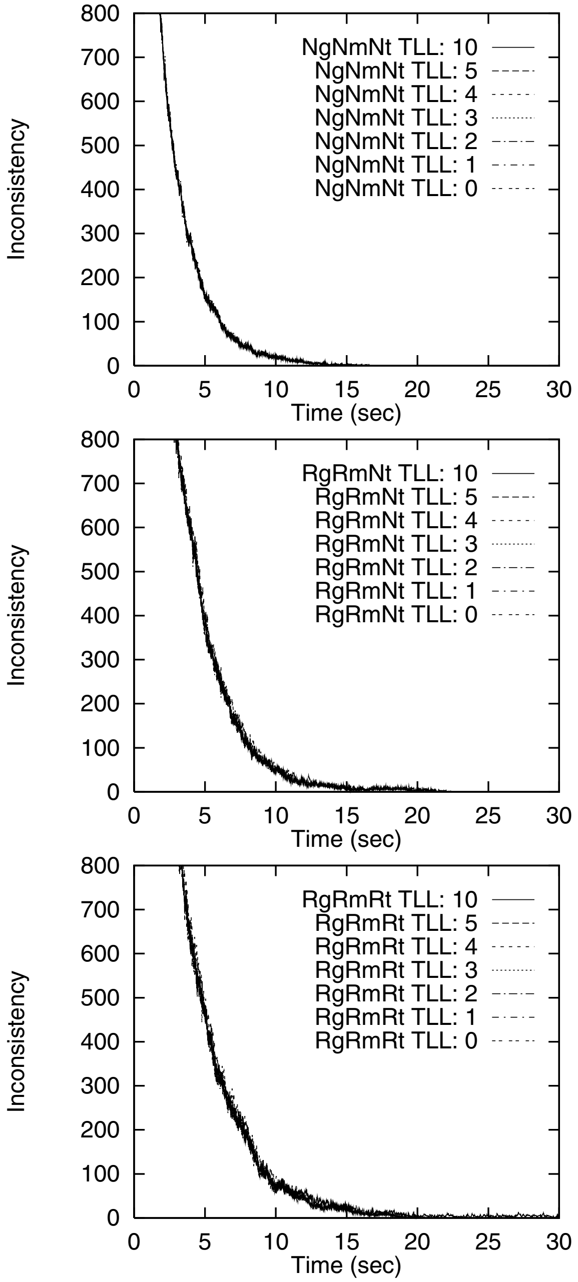


Fig. 2.15. Results for Tabu-List Lengths of 0, 1, 2, 3, 4, 5, and 10

stored in a first-in-first-out list and blocked for another selection as long as it is a member of the list.

Applying tabu lists proves absolutely pointless. Even for the nonrandomized NgNmNt version, it makes no difference.

2.6 Extending the Constraints

The constraints' improvement heuristics can be created in various ways. The effect of heuristics with more domain knowledge and the inclusion of more aggressive heuristics are studied in the following subsections.

2.6.1 More Knowledge

The experiments in Sect. 2.5.1 showed that the ARC's heuristic is not very powerful. It would therefore seem advisable to consider other heuristics. For example, tasks should obviously be packed quite tightly on a machine. The following ARC-H2 heuristic supports this feature. ARC-H2 is very similar to ARC-H1, but it makes the selection of the second interval in a different way: for the new position of the task, only two shifts are possible, the task either beginning at the beginning of the task's predecessor interval or ending at the end of the task's successor interval. The interval with less tasks is chosen.

The impact of the new heuristic is impressive. The best results are obtained using a probability of about 90 % for the ARC-H2 heuristic to be chosen, and 10 % for the ARC-H1 heuristic (see Fig. 2.16). Choosing the ARC-H2 heuristic more often causes a deterioration in the results.

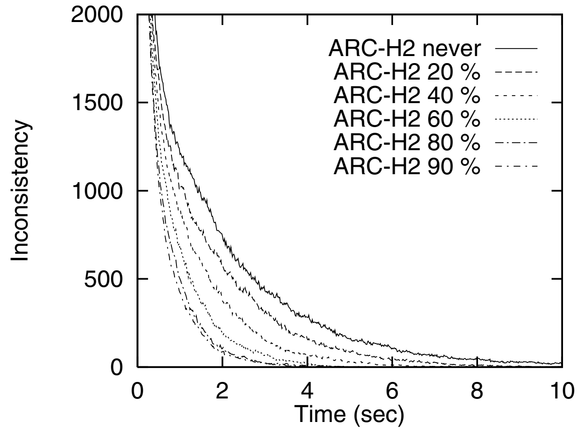


Fig. 2.16. Introduction of the New ARC's Heuristic

Figure 2.17 shows the impact of randomization, indicating the first interval's selection of the ARC-H2 heuristic by the letter *s* (always with a probability of 90 % for the ARC-H2 heuristic to be chosen). The addition of domain knowledge by the new heuristic strongly reduced the effect of randomization compared to that in Sect. 2.5.1.

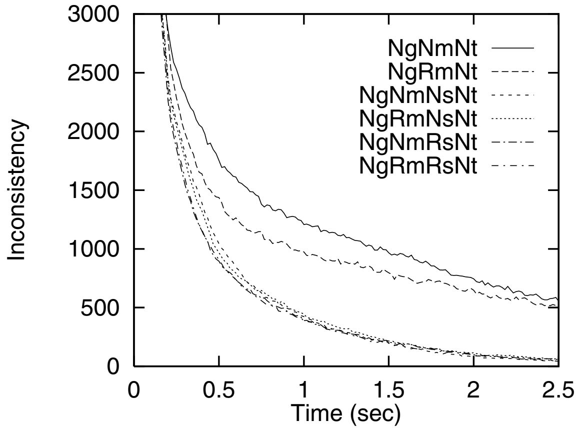


Fig. 2.17. Randomization with the New ARC's Heuristic

2.6.2 Aggressive Heuristics

The task constraint can also be extended. The current heuristic is very cautious, changing just one variable. After a couple of changes of task positions within a job, it may be useful to make a complete revision of the internal distance relations instead of repairing only one relation. A more aggressive heuristic can add to the former heuristic a recursive repair of all relations of the constraint whose inconsistency has been changed by the improvement (considering only variables that have not already been changed within the improvement step).

The effect is disappointing (Figure 2.18; using only the old ARC-H1 heuristic for the ARCs). Even a slight deterioration in the results is caused by activating the task constraint's new heuristic.

Cautious heuristics are often more appropriate than highly aggressive ones, the synthesis with other problem aspects being promoted by only slight changes of multi-constraint variables.

2.7 Conclusion

The presented combination of local search and constraint programming provides an important building block for our agents. Many other publications focus on problem-specific local search solutions. Improved efficiency is the

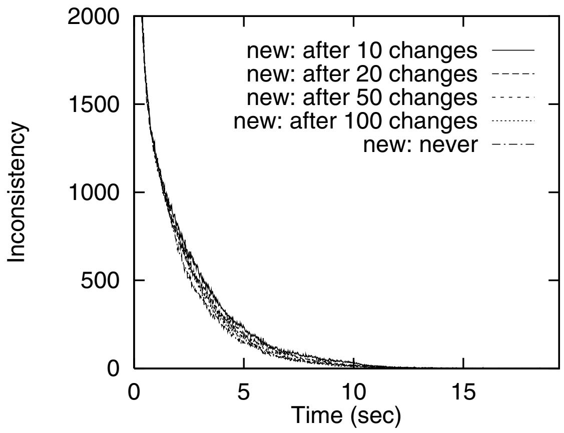


Fig. 2.18. Introduction of the New TC's Heuristic

main goal, generality often being disregarded. In contrast, our approach's *modular structure* of the constraints makes it *easy to vary, reuse and extend problem descriptions.*

Other authors have tackled the problem of combining local search with search frameworks on a more general level, too. This includes work on Boolean satisfiability problems like GSAT [78, 165] and Walksat [166], the processing of linear pseudo-Boolean constraint problems [185], and approaches for CSPs like coalition forming [86] and the well-known min-conflicts heuristic [121] with its extension and generalization by GENET [35]. The most important difference between our work and these approaches is the ability of the global constraints to exploit domain-specific information by including constraint-specific search control and representation knowledge. In contrast to low-level constraint programming approaches, which correspond rather to SAT- or OR-based approaches, the use of higher-level constraints is more in keeping with the basic intentions of constraint programming. Fine-grained constraints allow a wide application range, but the low-level problem decomposition also deprives the search process of most of the domain-specific knowledge.

The results of the presented case study indicate that the global constraint approach's *revision of a current state on a more global level* with an *inclusion of domain-specific knowledge* makes the search quite resistant to getting caught in local optima or on plateaus. Techniques to escape from local optima and plateaus, like randomization, random walks or tabu lists, had only a limited effect, and this for some decision points only. The advantage of these techniques further decreased with the inclusion of more domain knowledge.

The concept of global constraints was originally used for refinement search (e.g., by le Pape [111] and Puget and Leconte [152]). Transferring it to a local search context makes it possible to get an efficient and declarative handle on local search, while preserving features like reusability and maintenance.

3. Structural Constraint Satisfaction

In an environment that is only partially observable for an agent, it is not clear which objects exist, i.e., how many variables and resources of which kind are available. The closed-world assumption cannot be applied and we cannot restrict the planning process to a given set of state variables.

Furthermore, for a planning problem (and many other problem domains), numerous alternative structures are potentially valid for realizing a solution to a problem. Figure 3.1 gives an example of two alternatives/CSPs. The wizard can either cast a teleportation spell to get to a Halloween party and drink a magic potion to give himself an appropriate appearance for the party after he is teleported, or consult a map and walk while drinking the potion. This example shows that there are options not only with respect to the assignment of variables, but also to the graph structure (number and connection of the variables and constraints) itself.

Conventional constraint satisfaction is unable to handle structural alternatives, as constraint satisfaction problem formulations are static. There is a given set of constraints and variables, and the specified structure does not change. For the task of generating a plan, though, there are alternative CSP structures and the search for the structure of the CSP must be part of the satisfaction process – a **structural constraint satisfaction problem (SCSP)**.

The structural constraint satisfaction problem should not be confused with the dynamic constraint satisfaction problem (see [179] for a brief survey). Dynamic constraint satisfaction tries to revise a variable assignment with *given* changes to the constraint graph, e.g., finding a revised solution if a solution of $\{A = 1, B = 2, C = 3\}$ has been computed for a CSP that includes a constraint $A + B = C$, and this constraint is exchanged by $A + B = 2 \times C$. Dynamic constraint satisfaction does not include graph changes as part of the search.

In contrast to constructive approaches for building a graph, an SCSP adopts the constraint programming approach and defines solutions by specifying correctness tests. These test – so-called **structural constraints** – are restrictions on admissible constraint graphs. The basic satisfaction idea is to iteratively change a graph (starting with an empty graph) based on an

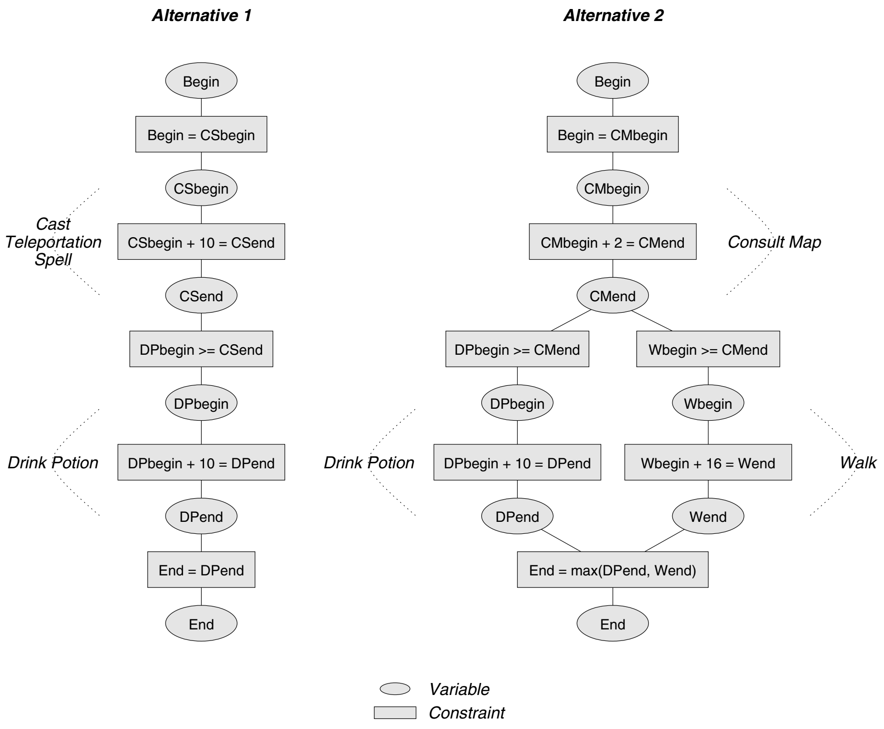


Fig. 3.1. Plan Alternatives

algebraic graph grammar, which is generically produced using the problem definition.

The presentation of the SCSP approach will be done by way of an example of an extended job-shop scheduling scenario, where a task can be executed with an arbitrary number of breaks in between. It is hardly possible to express this with a conventional CSP because the formulation would require a variable number of variables for the subtasks' start times and durations as well as a variable number of constraints.

Section 3.1 gives a brief introduction in graph grammars. This concept is extended by structural constraints in Sect. 3.3. The extension is used in Sect. 3.4 to formulate structural constraint satisfaction problems. Sections 3.5 and 3.6 show how to use the SCSP formulation to generically create a graph grammar with additional structural constraints to prevent from redundancies. The concept of structural constraint satisfaction is combined with the previous chapter's approach of global constraints for local search in Sect. 3.7.

3.1 Graph Grammars

This section provides a more or less informal introduction in algebraic graph grammars (see [49, 79, 158] for a detailed overview).

Algebraic graph grammars are a generalization of Chomsky grammars. A graph signature *GSig* consists of the sorts of vertices V, edges E, and a label alphabet L. The operations of *GSig* provide source and target vertices for every edge, $s, t : E \to V$, and associate a label to every vertex and edge, $l_v : V \to L$ and $l_e : E \to L$. Figure 3.2 shows an example graph.

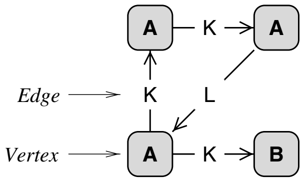


Fig. 3.2. Graph Grammars: A Graph

A match m of graph g_1 to graph g_2 is a total graph morphism that maps the vertices and edges of g_1 to g_2 such that the graphical structure and the labels are preserved. We use injective matches only.

A production P is a partial morphism between a left-hand side P_l and a right-hand side P_r, which provides information about which elements are preserved, deleted and created in the case of an application of the production. The identity of objects is marked by appended identifiers like `:1` (see Fig. 3.3). A production is applicable to a graph g, if there is a match of P_l to g. Figure 3.3 shows an example of a production. A general vertex that can match any vertex is graphically depicted by a flat ellipse with a dotted outline. A general edge that can match any edge is graphically depicted by a dotted line.

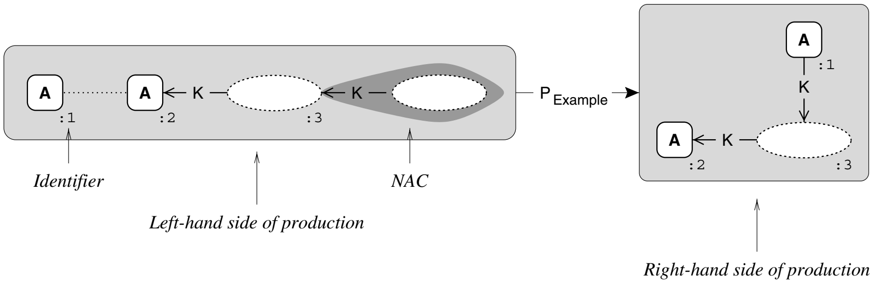


Fig. 3.3. Graph Grammars: A Production

A derivation g_2 of g_1 is the so-called *push-out* graph of an application of an applicable production P. The new graph g_2 is similar to g_1, but the elements of P_r that are not in P_l are added, and elements of P_l that are not in P_r are deleted. We formulate a general requirement for the deletion of vertices as it would be unclear how to proceed with dangling edges:

Requirement req_d: A production can specify the deletion of a vertex only if the vertex has no edge to another vertex.

Figure 3.4 shows how the graph of Fig. 3.2 can be transformed using the production of Fig. 3.3.

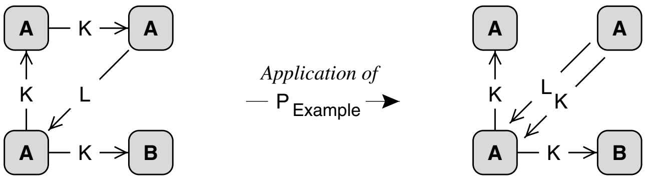


Fig. 3.4. Graph Grammars: A Derivation

The application of a production may also require application conditions. A negative application condition (NAC) is a total morphism $C : P_l \to n$ that is satisfied for a match $L : P_l \to g$ such that there is no morphism $G : n \to g$ such that $G \circ C = L$. An NAC is represented by a convex dark area (e.g., in the production of Fig. 3.3). A positive application condition (PAC) is a total morphism $C : P_l \to p$ that is satisfied for a match $L : P_l \to g$ if there is a morphism $G : p \to g$ such that $G \circ C = L$. A PAC is represented by a convex light area (e.g., in left-hand side of Fig. 3.16). For multiple application conditions, such as in production $P_{Machine_e}$ of Fig. 3.15, the conjunction of the conditions must hold.

Graph grammars can be used as a mechanism for describing potential neighbor states in the search space. This neighborhood is composed of all possible direct derivations of the current graph. The whole search space, such as the Cartesian product of all variables' domains in conventional CSPs, can be described by the corresponding graph language, with the empty graph as start graph.

3.2 Graph Elements for SCSPs

A variable of the CSP is represented by a vertex with the label *Variable*. It is graphically depicted by a circular vertex (see Fig. 3.5).

Constraints could be represented by edges, but there are constraint types that allow a variable number of variables to be included. These constraint types will be called *extensible constraints* (C_e) in contrast to *nonextensible* *constraints* (C_n), such that $C_e \cup C_n = C$. As the number of variables incorporated for extensible constraints may vary throughout the search, there must be a simple mechanism to include/exclude variables for constraints. Hence, a constraint is also represented by a vertex, new edges to variables being added in order to incorporate them. A constraint vertex's label corresponds to the type of the constraint. A constraint is graphically depicted by a rectangular vertex.

An SCSP allows the existence of so-called *object constraints*. These constraints do not restrict the variables' values, but provide structural context information. For example, for the job-shop scheduling example of the chapter's introduction, it must be known which two *Start* and *Duration* variables together form a SubTask object. Otherwise, a Machine constraint designed to check if the included SubTasks overlap might combine *Start* and *Duration* variables of different SubTasks for the check. Object constraints act as a kind of structural broker between variables and conventional constraints. They are represented by a rectangular vertex with a dashed outline. Object constraints can be nonextensible as well as extensible, $O_e \cup O_n = O$, $C \cap O = \emptyset$. Constraints of C can be connected to variables and object constraints, whereas object constraints can be connected to constraints of C as well.

Edges are used to connect vertex elements. The role/position of a variable (or constraint when speaking about objects) within a constraint is often very important. Thus, an edge's label and direction can be used to express its role/position[1]. An edge's direction is indicated by an arrow and the label is displayed in the edge's middle (*NoLabel* if omitted).

Figure 3.5 shows an example graph with an extensible conventional constraint Machine, an extensible object constraint Task, two nonextensible object constraints SubTask, three extensible conventional constraints Sum and a nonextensible conventional constraint Less. The Machine constraint ensures that the *Start* and *Duration* variables of all connected SubTasks (via Tasks) do not produce an overlap. The Sum constraint restricts the sum of ($\circ \to \Box$)-connected variables to equal a ($\Box \to \circ$)-connected variable. The Less constraint that restricts the ($\circ \to \Box$)-connected variable to be less than the ($\Box \to \circ$)-connected variable.

3.3 Structural Constraints

Structural constraints could be freely defined automata to test graphs for specific properties. However, to enhance computability, we use a stricter convention here.

A conventional CSP's constraint correlates domain values. In contrast, a structural constraint correlates subgraphs. A conventional constraint's application point is defined by the problem formulation, whereas a structural

[1] Labels could just as well be expressed by a special object constraint in between.

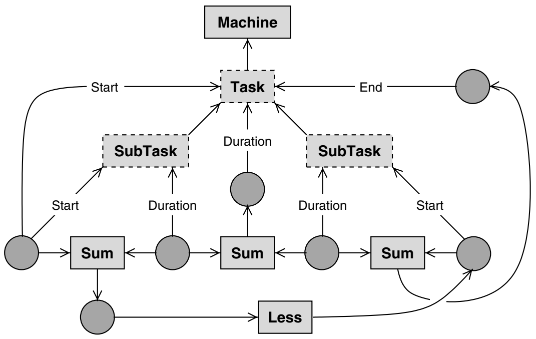


Fig. 3.5. An Example Graph

constraint's application point is not clear in advance. Thus, a structural constraint needs a matching part that is equal to the left-hand side of a production rule (*docking part* S_d).

A conventional constraint is true as long as there is at least one tuple of possible variable assignments. There may be a few possible structures to be accepted by a structural constraint as well. Thus, structural constraints do not have only one right-hand side, like a production, but a set of alternatives (*testing part* S_t). These alternatives have a testing nature and are not used for pushouts like the production's right-hand side. Because of this, there may be application conditions not only for the docking part, but also for the testing part's alternatives.

If the docking part S_d of a structural constraint $S = (S_d, S_t)$ matches the constraint graph, an alternative of the testing part S_t has to match too. A graph g is structurally consistent iff there exists a morphism $T : a \to g$, $a \in S_t$ for every structural constraint S and every possible match $D : S_d \to g$ such that $T \circ (S_d \to a) = D$.

Figure 3.6 shows the restriction that a SubTask must be the only beginning of a Task's temporally ordered list of SubTasks (first alternative), at the end (second alternative), somewhere in the middle (third alternative) or the only SubTask (fourth alternative). Further structural constraints ensure that a SubTask has at most one successor in the Task's list (see Fig. 3.7), that a Task has at least one SubTask and is processed on exactly one Machine (see Fig. 3.8) and that the Sum of the SubTasks' *Duration*s equals the Task's *Duration* (see Fig. 3.9).

Heckel and Wagner [85] introduce so-called *consistency conditions* that are equal to structural constraints, with a testing part consisting of one alternative only. These can be directly transformed into semantically equivalent application conditions of productions.

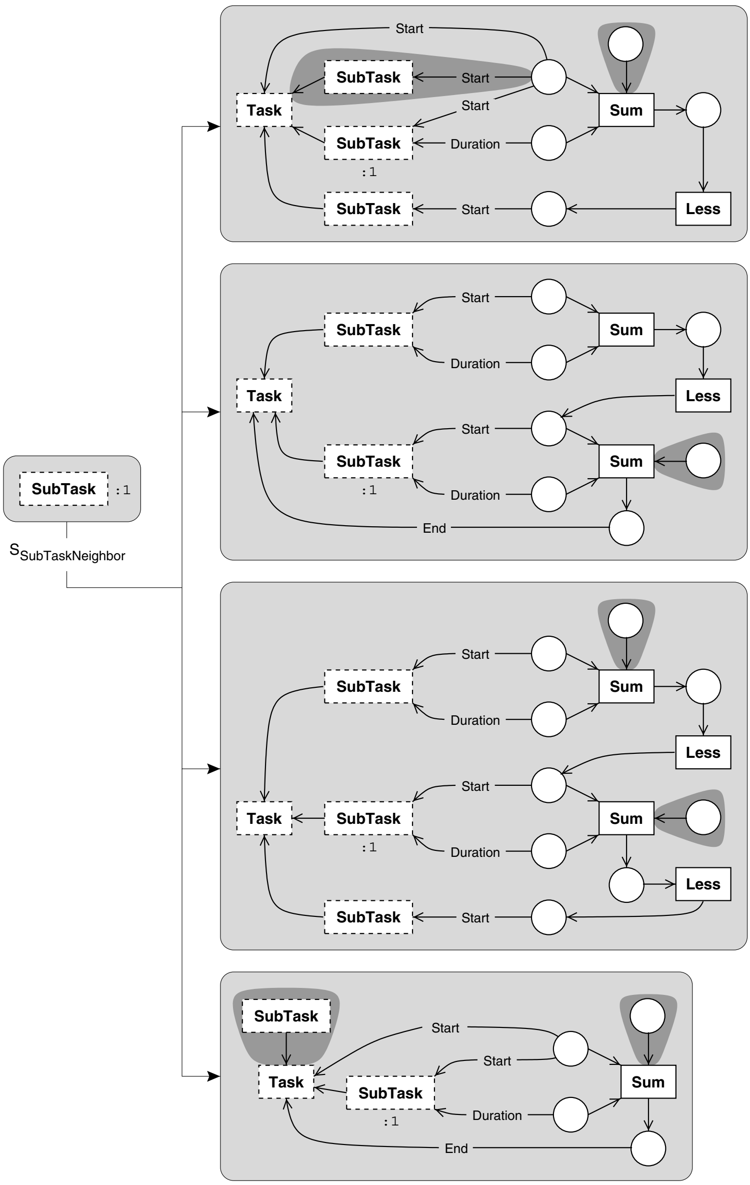


Fig. 3.6. The Structural Constraint SubTaskNeighbor

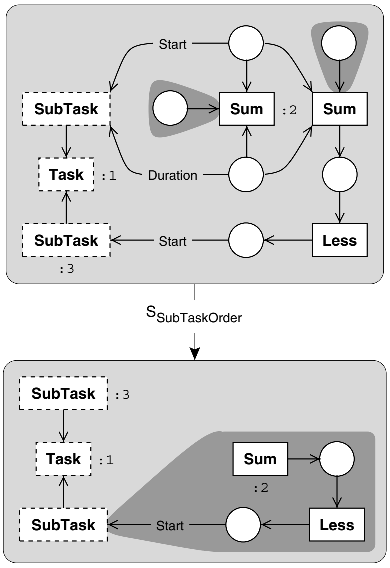


Fig. 3.7. The Structural Constraint SubTaskOrder

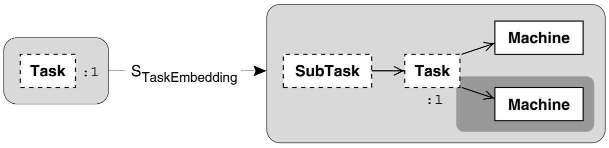


Fig. 3.8. The Structural Constraint TaskEmbedding

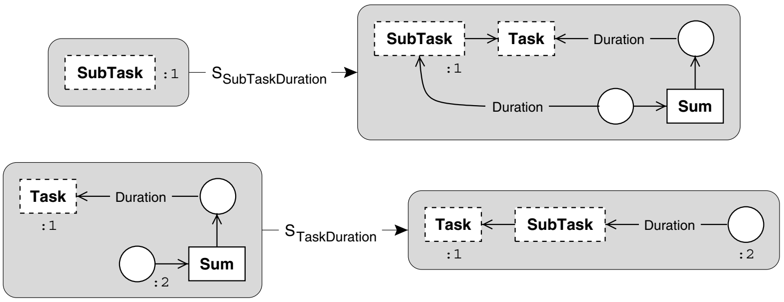


Fig. 3.9. The Structural Constraints SubTaskDuration and TaskDuration

3.4 Structural Constraint Satisfaction Problems

A solution to an SCSP is a constraint graph with an arbitrary number of variables, conventional (or object) constraints and connecting edges, such that the graph satisfies all structural constraints. Thus, the basic ingredients to formulate an SCSP are a set of types of conventional (or object) constraints and a set of structural constraints. But in most cases there will be much less allowed constellations of conventional (or object) constraints than inconsistent ones. This means that – at least for the direct neighborhood of a constraint – a constructive specification is more appropriate than a huge number of structural constraints. Because of this, our SCSP specification will include generic constellations alternatives for the direct embedding of a conventional (or object) constraint and minima and maxima for the direct embedding of extensible constraints. This is comparable with conventional CSPs, where a variable's domain is also mostly represented by enumeration mechanisms with minima and maxima instead of constraints.

> A structural constraint satisfaction problem $SCSP = (\mathcal{CD}, \mathcal{S})$ consists of a tuple of sets of constraint descriptions $\mathcal{CD} = (\mathcal{C}_n, \mathcal{C}_e, \mathcal{O}_n, \mathcal{O}_e)$ and a set of structural constraints $\mathcal{S}$. The constraint descriptions of $\mathcal{C}_n$ and $\mathcal{O}_n$ are pairs (c, p_{base}) with a nonextensible conventional (or object) constraint c and its embedding graph p_{base}. The constraint descriptions of $\mathcal{C}_e$ and $\mathcal{O}_e$ are 4-tupel $(c, p_{base}, E, p_{max})$ with an extensible conventional (or object) constraint c, its minimal embedding graph p_{base}, a set of extension graphs E and the constraint's maximal embedding graph p_{max}.

An embedding graph shows the constraint with all its directly connected neighbor vertices. If an extensible constraint has no maximal embedding, p_{max} is the empty graph. An extension graph shows the constraint connected to the vertices that can be added in one step. Figure 3.10 shows the residual components of the example SCSP. To show the effects of maximal embedding graphs, we introduce a maximum of two Tasks for a Machine.

There are some requirements that are induced by the construction of the search space in the following section (fulfilled for the presented example SCSP):

- Nonextensible constraints are not allowed to appear in graphs of other constraints, as the addition and deletion productions of constraints partly incorporate their graphs (to satisfy req_{ne}).
- Constraint-usage cycles are not allowed for the p_{base} graphs of extensible constraints, e.g., that p_{base_A} includes a B constraint and p_{base_B} includes the A constraint (to satisfy req_e).
- If the p_{max} graph of an extensible constraint is non-empty, no sequence of productions that is applied to the constraint's p_{base} graph can produce a graph that includes the p_{max} graph without previously producing the p_{max} graph (to satisfy req_{max}).

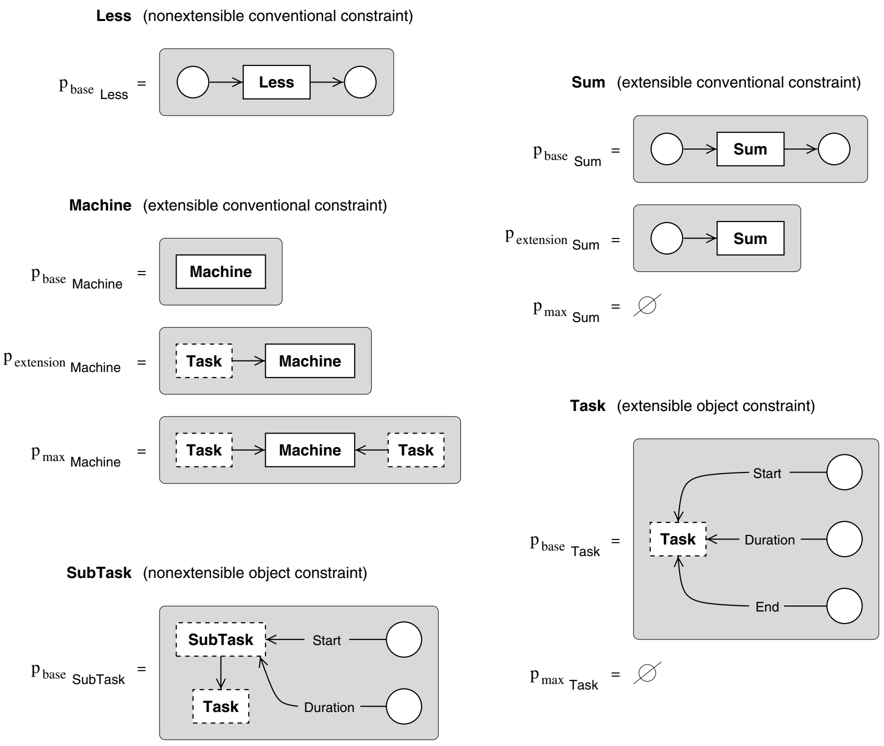


Fig. 3.10. Components of the Example SCSP

These requirements can be relaxed by using more sophisticated ways to generate the search space. However, this is not necessary for the work presented here.

3.5 Generating the Search Space

For conventional constraint satisfaction, values are assigned to variables. This builds the search tree/space with potential solutions. In contrast, structural constraint satisfaction uses productions to create constraint graphs. Figure 3.11 shows an example of a search space for an SCSP.

It must be guaranteed that all valid constraint graphs can be constructed by the productions. For refinement search, an empty start graph can continuously be expand toward a possible constraint graph by addition productions for the single elements (see [130] for structural constraint satisfaction approaches for refinement search). For local search techniques, these addition productions are not sufficient as any state must be reachable from any other state and not only from the empty start graph. This requirement can be satisfied by the introduction of further deletion productions:

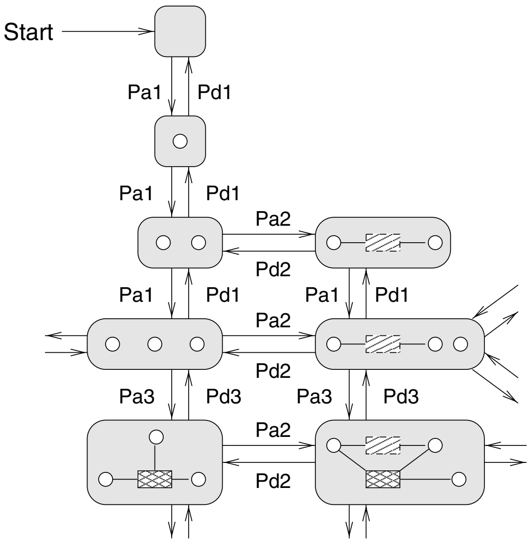


Fig. 3.11. An Example of a Structural Search Space

> **Requirement** req_p: The effect of an addition production can be retracted by a directly following application of a corresponding deletion production.

Analogously to the constraint checking in conventional constraint satisfaction, an SCSP's structural constraints have to be checked during search. The productions, together with the structural constraints, provide a tool to allow search to generate and verify a solution. However, the search space is potentially infinite and the support of preemptive alternative reductions and domain-specific search knowledge about which production to apply and where to apply it on is very important (like propagation/consistency methods and variable- and value-ordering heuristics in conventional constraint satisfaction). This is elaborated in Sect. 3.7.

The following subsections provide rules to create productions that can add/delete all potential graph elements. In the same way as values that are not in the domains of variables are not considered in conventional constraint satisfaction, graphs that violate the SCSP's embedding and extension graphs are implicitly prevented by the productions.

3.5.1 Productions for Variables

Addition. One production ensures that it is always possible to add further variables. The production is shown in Fig. 3.12 (production P_{v_a}).

Deletion. The production for the deletion of a variable (production P_{v_d} in Fig. 3.12) requires that the variable is not connected to any constraint (to satisfy req_d). This is ensured by the NAC. The NAC cannot endanger req_p as it is not possible that there are edges to the variable directly after the application of P_{v_a}.

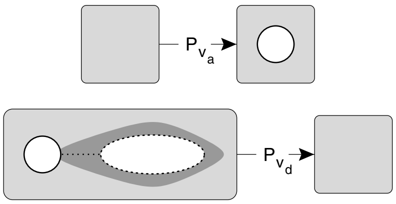


Fig. 3.12. Productions for Variables

3.5.2 Productions for Nonextensible Constraints

Addition. To guarantee that an extensible constraint cannot exceed its maximal configuration p_{max}, it must be ensured that edges to an extensible constraint with a nonempty p_{max} graph can only be added if the constraint's p_{max} graph is not exceeded. An addition production can include NACs with the involved constraints' p_{max} graphs to ensure that a maximal configuration is not already existent. This requires that

> **Requirement** req_{max}: If the p_{max} graph of an extensible constraint is nonempty, any derivation sequence of productions that is applied to the constraint's p_{base} graph cannot produce a graph that includes the p_{max} graph without producing the p_{max} graph before.

The p_{base} specification states the only consistent way to connect a nonextensible constraint with variables. Therefore, a whole p_{base} structure can be established by an addition production at once. Multiple similar nonextensible constraints between the same elements can be prohibited by an NAC for the addition production. This is not a vital structural constraint but saves from redundancy.

In conclusion, there must be one addition production per nonextensible constraint, which is constructed in the following way:

> **Construction** $P_{nonextensible_a}$: The right-hand side of the addition production is equal to the constraint's embedding graph p_{base}. The left-hand side of the production contains the vertices of the right-hand side without the constraint itself, and an NAC that contains the right-hand side without the vertices that are connected to the constraint. In addition, the left-hand side includes an NAC for each included extensible constraint to prevent from exceeding this constraint's p_{max} graph. These NACs consist of a constraint's maximal

embedding graph p_{max} such that the constraint is unified with the constraint of the left-hand side (NAC without the constraint itself).

Production P_{Less_a} in Fig. 3.13 shows the addition production for the Less constraint.

The left-hand side of the addition production requires that the other vertices of p_{base} are already existent. To prevent from deadlocks, it must be possible to create the other vertices of p_{base} independently of the nonextensible constraint. This is enforced by

Requirement req_{ne}: The occurrence of a nonextensible constraint is only allowed in its addition and deletion production.

This requirement also ensures that the p_{base} specification is always satisfied, as no production can connect or disconnect vertices from the constraint.

Deletion. To guarantee that an extensional constraint in the graph cannot be changed by productions below its minimal configuration p_{base}, it must be ensured that

Requirement req_{min}: Edges to an extensible constraint can only be deleted (without deleting the constraint itself), if at least the constraint's p_{base} graph is preserved.

There must be one deletion production per nonextensible constraint. It withdraws the addition of the corresponding $P_{nonextensible_a}$ production:

Construction $P_{nonextensible_d}$: The left-hand side of a deletion production is equal to the embedding graph p_{base}. The right-hand side of the production contains the vertices of the left-hand side without the constraint itself. In addition, the left-hand side includes a PAC for each included extensible constraint to prevent from falling below this constraint's p_{base} graph. These PACs consist of a constraint's embedding graph p_{base} such that the constraint is unified with the constraint of the left-hand side (PAC without the constraint itself).

Production P_{Less_d} in Fig. 3.13 shows the deletion production for the Less constraint.

req_d can be neglected for the deletion productions of nonextensible constraints, as no additional vertex can be connected to the constraint because of req_{ne}. req_p is satisfied, as $P_{nonextensible_d}$ is the reversal of $P_{nonextensible_a}$ without its NAC and as the PACs of $P_{nonextensible_d}$ cannot endanger the applicability as the existence of the p_{base} graphs is ensure by req_{min}.

3.5.3 Productions for Extensible Constraints

Addition. Extensible constraints cannot be added in one step like the nonextensible constraints. Only the extensible constraint's p_{base} graph can be added at once, as this is the minimal structure. Thus, the addition production

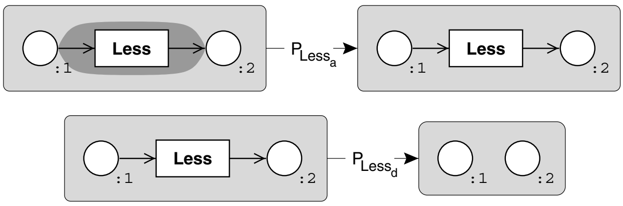


Fig. 3.13. Productions for Nonextensible Constraints (Examples)

for an extensible constraint is similar to the one of nonextensible constraints, but without the NAC to avoid redundancy (for an avoidance from redundancy see Sect. 3.6):

> **Construction** $P_{extensible_a}$: The right-hand side of the addition production is equal to the constraint's embedding graph p_{base}. The left-hand side of the production contains the vertices of the right-hand side without the constraint itself. In addition, the left-hand side includes an NAC for each included constraint. An NAC consists of a constraint's maximal embedding graph p_{max} such that the constraint is unified with the constraint of the left-hand side (NAC without the constraint itself).

Production P_{Sum_a} in Fig. 3.14 shows an example for the Sum constraint.

The left-hand side of the addition production requires that the other vertices of p_{base} are already existent. To prevent from deadlocks, it must be possible to create the other vertices of p_{base} independently of the extensible constraint. This is enforced by

> **Requirement** req_e: Constraint-usage cycles are not allowed for the base embedding graphs of extensible constraints.

Deletion. There must also be one deletion production per extensible constraint. It withdraws the addition of the corresponding $P_{extensible_a}$ production. The production is generated according to the one of a nonextensible constraint, but must contain an NAC to forbid edges to further vertices (to satisfy req_d):

> **Construction** $P_{extensible_d}$: The left-hand side of the deletion production is equal to the embedding graph p_{base}. The right-hand side of the production contains the vertices of the left-hand side without the constraint itself. The left-hand side contains an NAC that has a general vertex with a general edge to the constraint.

Production P_{Sum_d} in Fig. 3.14 shows an example for the Sum constraint.

req_p is satisfied, as $P_{extensible_d}$ is the reversal of $P_{extensible_a}$ and the NAC cannot endanger the applicability as it is not possible that there are edges to the constraint directly after the application of $P_{extensible_a}$. $P_{extensible_d}$ cannot endanger the req_{min} requirement for the other constraints of p_{base} because of req_e.

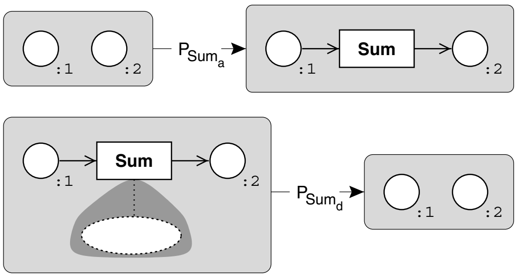


Fig. 3.14. Productions for Extensible Constraints (Examples)

3.5.4 Productions for Constraint Extensions

Extension. Every possible extension of an extensible constraint can be added in one step. Multiple similar extensions between the same elements can be prohibited by further NACs for the production. These are not vital structural constraints but save from redundancy.

There must be one production for every possible extension of an extensible constraint:

> **Construction** $P_{extensible_e}$: The right-hand side of the production is an extension graph of E. The left-hand side is created by the vertices of the right-hand side, an NAC for each edge of the right-hand side, and, if p_{max} is not empty, an NAC consisting of the constraint's maximal embedding graph p_{max} such that the constraint is unified with the constraint of the right-hand side (NAC without the constraint itself). In the same way as the last NAC, further NACs have to be introduced for all other constraint vertices of the extension graph to prevent from exceeding the constraints' p_{max} graphs.

Production $P_{Machine_e}$ in Fig. 3.15 shows an example for an extension of the Machine constraint.

Reduction. For every extension of a constraint, there must be a production to remove the extension. The requirement req_{min} can be satisfied by providing an additional PAC with the p_{base} graph for each constraint:

> **Construction** $P_{extensible_r}$: The production's left-hand side is similar to the extension graph of E, including a PAC that contains the p_{base} graph, such that the constraint is unified with the constraint of the extension graph (PAC without the constraint itself). In the same way as the PAC, further PACs have to be introduced for all other

constraint vertices of the extension graph to prevent from falling below the constraints' p_{base} graphs. The right-hand side is the left-hand side without the PACs and the edges.

Production $P_{Machine_r}$ in Fig. 3.15 shows an example for the reduction of the Machine constraint.

req_p is satisfied, as $P_{extensible_r}$ is the reversal of $P_{extensible_e}$ without its NACs, and as the PACs cannot endanger the applicability as the existence of the p_{base} graphs is ensure by req_{min}.

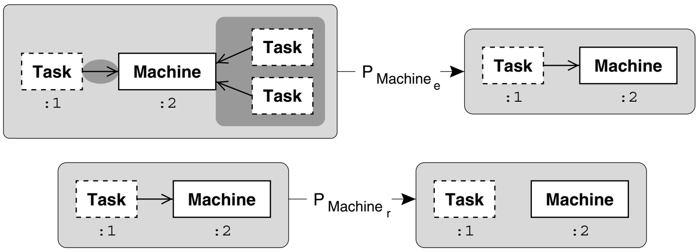


Fig. 3.15. Productions for Constraint Extensions (Examples)

3.6 Avoiding Redundancy

The addition of nonextensible constraints prevents redundant constraints by means of a corresponding NAC in $P_{nonextensible_a}$-productions. This avoidance of redundancy is not ensured for extensible constraints. Generic structural constraints can overcome this problem.

A production for extending an extensible constraint $P_{extensible_e}$ adds further edges to the graph. Extensible constraints of the same type must differ by involving at least one other (or additional) element. There must be a structural constraint $S_{extensible}$ for each extensible constraint type, docking at each pair of potentially redundant constraints, and having all possible distinctive features as test alternatives.

Potentially redundant constraints are two constraints of the same type that are connected to the same elements according to the base graph p_{base}:

Construction $S_{extensible}$ – **docking part**: The docking part of the structural constraint is created by two p_{base} graphs, where the corresponding vertices (without the constraints themselves) are unified. To avoid multiple redundant structural constraint instances per potentially redundant constraint pair, all but the constraints themselves are a PAC.

Possible distinctive features are vertices that are connected to one constraint but not (in the same way) to the other:

> **Construction** $S_{extensible}$ – **testing part**: There are two alternatives per unique edge (label; direction with respect to the constraint – toward it or away from it) that is included in the constraint's extension graphs. Each of the two alternatives consists of a graph with the two constraints of the docking part and an additional general vertex. Between each constraint and the general vertex is an edge corresponding to the unique edge. In the one alternative, the edge to the first constraint is an NAC; in the other alternative, the edge to the second constraint is an NAC.

Figure 3.16 shows an example of a Subset constraint that forces the set of (○→□)-connected variables to be a subset of the set of (□→○)-connected variables. The p_{base} graph of a Subset constraint includes two variables (○→□→○), and there are two possible extensions corresponding to the two possible variable connections.

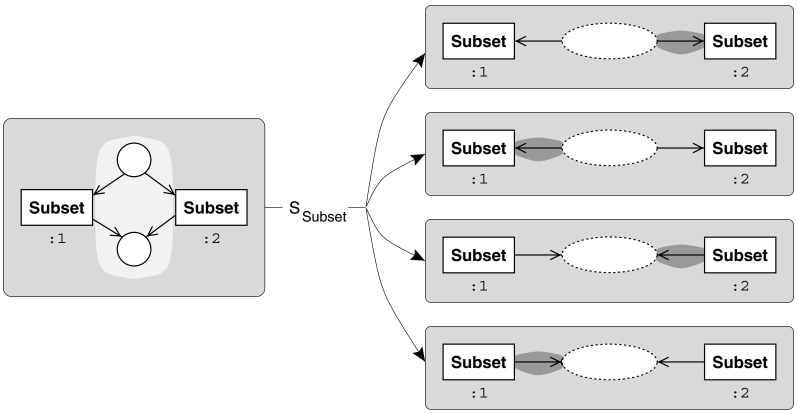


Fig. 3.16. Avoidance of Redundant Subset Constraints

3.7 Combination with the Global Constraint Approach

Given an SCSP, valid constraint graphs must somehow be constructed. This section adopts the local-search approach with global constraints to handle structural constraint satisfaction. In this context, the local-search approach is extended to handle optimization tasks.

3.7.1 Application of Structural Constraints

If the graph is changed by a production, the graph's consistency must be verified again. It would be very costly to test each time for all matches of possible structural constraints. Instead, *instances* of structural-constraint *types* are memorized. These instances stay matched to a certain part of the constraint graph. Then, the graph must only be reverified with respect to the changes.

Whenever new elements are added by a production, there must be new structural-constraint instances for all possible constraints' docking-part matches that include the new elements. Since there may be negative application conditions in the docking parts of existing structural constraints, those with an unsatisfied negative application condition due to the new elements must be excluded.

Whenever elements are deleted by a production, all structural-constraint instances that matched these elements with their docking part must be deleted. If the deleted elements were part of a PAC in the docking part, the constraint instances must only be deleted if no other elements are available to satisfy the PAC. The deletion of elements may also mean that NACs of potentially applicable constraints' docking parts are no longer applicable. Structural-constraint instances must be added in these cases.

3.7.2 Types of Global Constraints

The approach of using global constraints for local search covers only the satisfaction of conventional global constraints. Besides conventional global constraints, there are now global object and structural constraints as well. Conventional and object constraints are added/deleted by productions, a check being necessary after each application of a production to decide whether to add/delete global structural constraints (see Sect. 3.7.1). After applying a production, the inconsistency of affected structural constraints must be updated.

Extensible global (conventional or object) constraints must feature additional update functions to integrate/disintegrate variables and object constraints.

Global Conventional Constraints. The heuristics of global conventional constraints can now additionally change the graph structure by applying a production or a sequence of productions. For example, the graph contains a structure like the second test alternative of the structural constraint SubTaskNeighbor (see Fig. 3.6), and the Less constraint's heuristic would decide on a structural change because it is not able to enforce the variables' order. In consequence, the Less constraint might apply the production P_{Less_d} to delete itself.

Global Structural Constraints. The global structural constraints are very similar to global conventional constraints. One difference is that their costs are not associated with variable assignments, but with the existence or nonexistence of graph elements. Thus, their heuristics can only apply productions. A change of variable values is not allowed (apart from an initial value assignment that they have to provide for created variables).

For example, after deletion of the LESS constraint, the global structural constraint that corresponds to the structural constraint SUBTASKNEIGHBOR (see Fig. 3.6) may become inconsistent. The structural constraint's heuristic might decide to satisfy its fourth alternative because the second alternative has become inconsistent. This can be accomplished by using the TASK's *Start* variable as the SUBTASK's *Start* variable, and by deleting all other SUBTASKs of the TASK.

Global Object Constraints. Global object constraints are only a reference structure and do not have cost or improvement functions. Their role is to maintain a correct linking of conventional constraints and variables such that the variable updates can be passed correctly and the conventional constraints are aware of the variables' structural connection.

A conventional constraint must inform connected object constraints about the variables it is interested in, as not all connected variables may be important. For example, a MACHINE needs information about the involved *Start* and *Duration* variables of a TASK's SUBTASKs, but not about the TASK's own variables.

3.7.3 Global Search Control

If a structural constraint becomes unsatisfied by a graph change, the global search control must ensure a correct graph structure before proceeding, as conventional constraints cannot handle inconsistent structures. Thus, the global search control always selects a structural constraint to improve the current state if there is an inconsistent structural constraint (see Fig. 3.17).

3.7.4 Goal Optimization

So far, there has only been a search for a consistent plan. But we wish to go beyond satisfaction and optimize the agent's plan according to given goals. These goals may have a conventional as well as a structural nature. Thus, conventional and structural constraints can have a second cost contribution for a second cost function (*goal function*).

For conventional constraints, it is fairly obvious how the goal function can be influenced, e.g., by the completion time of a task, the workload of a resource or monetary resource costs. The contribution of structural constraints to the goal function focuses mostly on graph minimization, e.g., a

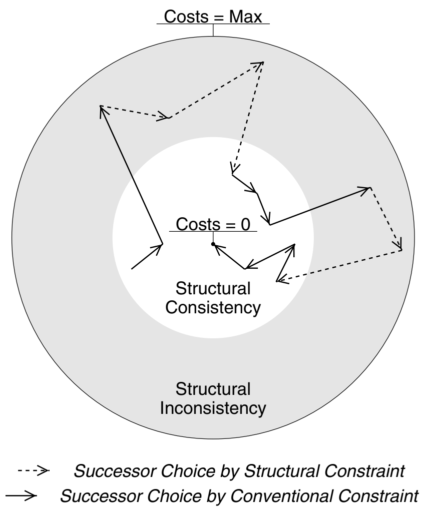

Costs = Max
Costs = 0
Structural Consistency
Structural Inconsistency
Successor Choice by Structural Constraint
Successor Choice by Conventional Constraint

Fig. 3.17. Exploration of the Search Space

structural constraint that forbids redundant structures or one that checks for unconnected object constraints.

The global search control handles this second cost function as well. The functions may compete such that an improvement of one function value may lead to a deterioration in the other function value. Thus, various options for the selection of a function for improvement should be customizable, e.g., alternating phases of a predefined length or with special abort criteria, or a general cost measure consisting of the addition of the single function values multiplied by static (or even variable) coefficients. A constraint that is selected for improvement is told whether to improve its goal or its cost function.

Figure 3.18 shows the revised components' interplay (cf. Fig. 2.2).

3.8 Conclusion

This chapter has provided the basic mechanism to handle arbitrary structures for an agent's behavior plan, enabling us to treat open-world problems with a potentially infinite number of objects. Being embedded in the general constraint programming paradigm, the framework is not restricted to a certain way of exploring the search space, e.g., according to an increasing plan length.

Combining conventional constraint satisfaction with structural requirements enables us to formulate and solve combinatorial search problems without explicitly giving the solution's structure. The SCSP approach follows the declarative constraint programming paradigm by stating only requirements for the solution and without including information on solution generation.

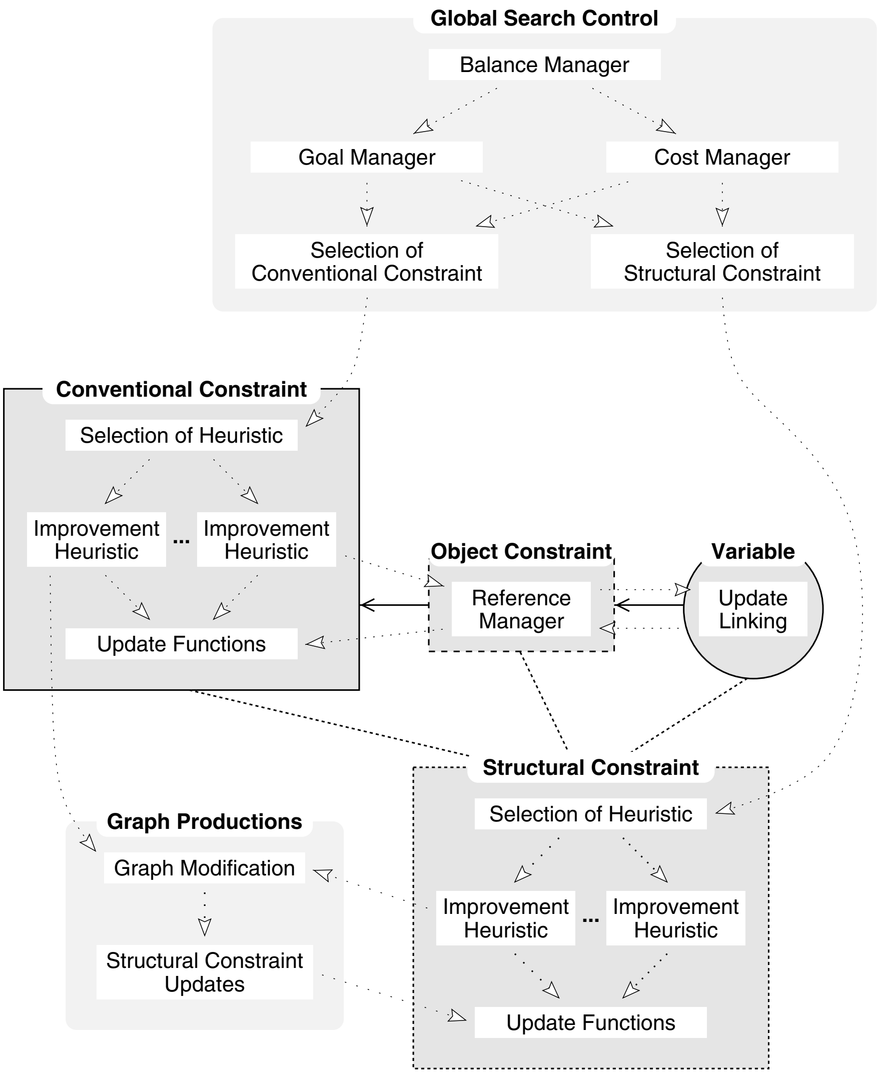


Fig. 3.18. Local Search for SCSPs

Productions can be deduced from the SCSP's embedding and extension graphs, which allow search to generate all potentially valid solutions. During search, the SCSP's structural constraints must be checked to ensure the graph's consistency.

SCSPs allow us to tackle a new class of problems by constraint programming techniques. An example for the need of structural alternatives is action planning, where it is not clear in advance how many and what kind of actions are needed and how they must be arranged. The same holds for configuration and design problems, where type and number of parts and the parts' relations must be determined as part of the search process.

The concept of structural constraint satisfaction contrasts with previous approaches that try to overcome the problem by considering maximal structures with deactivatable elements (e.g., [122, 137]). The use of maximal structures is useful in the case of only slightly variable structures and a known maximum. A formulation by an SCSP does not have these restrictions.

Composite CSPs [159] aim at a similar extension of the conventional constraint satisfaction paradigm as SCSPs. A composite CSP expresses subgraph alternatives in a hierarchical way. This allows optimized search guidance, but requires manual preprocessing to build the hierarchy. The creation of the hierarchy it is often problematic, as completeness and appropriate structure are not always obvious. This pre-structuring of the search space is similar to using a pre-defined static variable/value ordering within refinement search.

The combination of structural constraint satisfaction with the approach of global constraints for local search forms the basis for formulating and efficiently solving planning problems within the constraint programming framework.

4. The Planning Model

The previous chapters have provided basic specification and solving techniques that can be used to handle the agents' planning. This chapter introduces a planning model that makes use of these techniques.

The model focuses on resources. A resource (also called *state variable* or *fluent*)[1] is a temporal projection of a specific property's state, which may be subject to constraints such as preconditions and changes. Numerical as well as symbolic properties are uniformly treated as resources. For example, a battery's Power and the state of a Door are resources:

Power is [0: $t \in [0..5]$, $10 - 0.75 \times t$: $t \in [6..13]$, 0: $t \in [14..\infty[$],
Door is [Open: $t \in [0..45]$, Closed: $t \in [46..60]$, Unknown: $t \in [61..\infty[$].

4.1 The Model's Basics

This section introduces the model's basics. A formal specification will be given in the following section. The model's basic concepts can be grouped with respect to **actions**, **states** and higher-level **objects**.

4.1.1 Actions, Action Tasks, and Action Resources

The execution of an **action** (like Eat Peanut) includes **action task** subcomponents. These action tasks represent operations that are necessary to carry out the actions. Each of the action tasks utilizes an **action resource** for its execution. For instance, the action Eat Peanut requires action tasks on a Mouth and a Left Hand or Right Hand action resource. Figure 4.1 visualizes the assignment of action tasks to action resources.

It is forbidden for action tasks on the same action resource to overlap, as simultaneous executions of tasks would interfere with each other. For example, the agent is not allowed to talk and eat with his Mouth at the same time.

[1] *Resource* is the term commonly used in the CP/OR community and is used here because of the planning system's close relation to applications for resource allocation/optimization.

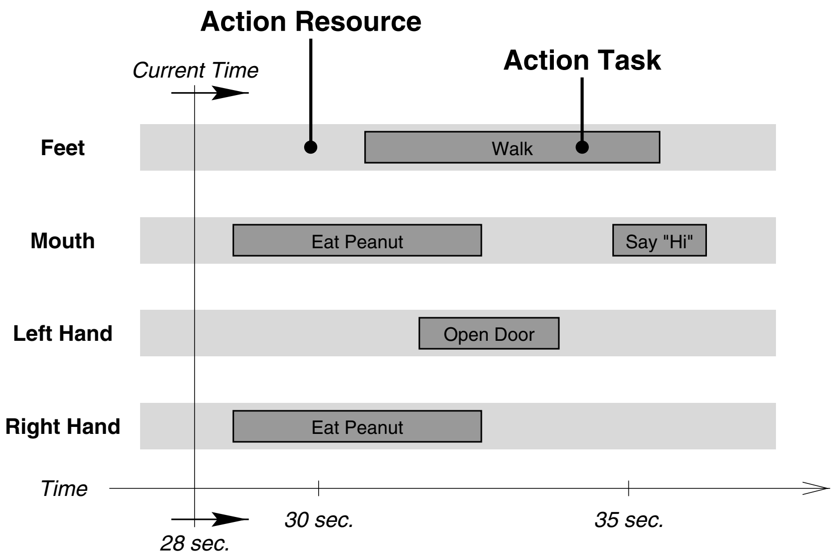


Fig. 4.1. The Assignment of Action Tasks to Action Resources

The tasks of an action are subject to action-specific conditions. For example, the action tasks of the action EAT PEANUT must begin and end at the same time, and the begin and end values must be four seconds apart.

4.1.2 State Resources, State Tasks, and Precondition Tasks

A **state resource** is similar to an action resource. It does not manage actively planned actions, but rather the development of a specific property of the environment or the agent itself. For example, an OWN PEANUT state resource with a Boolean assignment for any one time can provide information about the possession of a peanut (see Fig. 4.2).

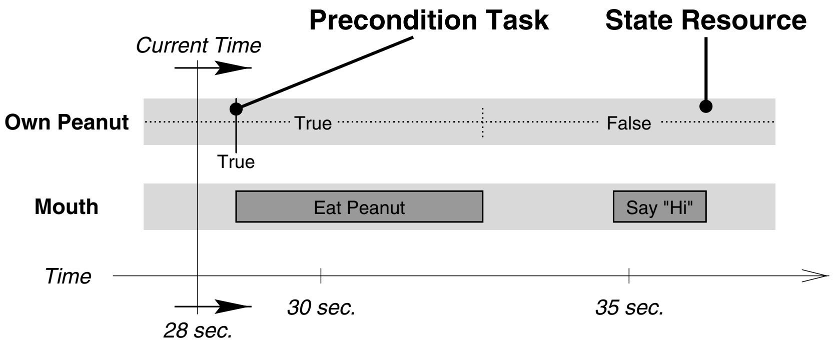


Fig. 4.2. A State Resource

The status of the state resources can restrict the applicability of actions. To execute the action Eat Peanut, it is first necessary to have a peanut. These relations are checked by **precondition tasks** of actions. A precondition task includes a state value (or value ranges) that must correspond with the state of a related state resource at a specific time.

The effects of actions are more complicated to realize, as multiple actions and events may have synergistic effects. For example, a state resource Hunger with assignments of natural numbers can be influenced by a beneficial action Eat Peanut and a detrimental Walk at the same time.

It is the job of **state tasks** to describe an action's effects. For instance, a state task of the action Eat Peanut is responsible for a decreasing **contribution** of -3 to the state resource Hunger during the action's execution (see Fig. 4.3). Each state resource has a specific **state mapping** that maps the contributions of the state tasks to values of the state resource's domain. In the case of the Hunger resource, the synergistic effect is a simple addition of the state tasks' single gradients.

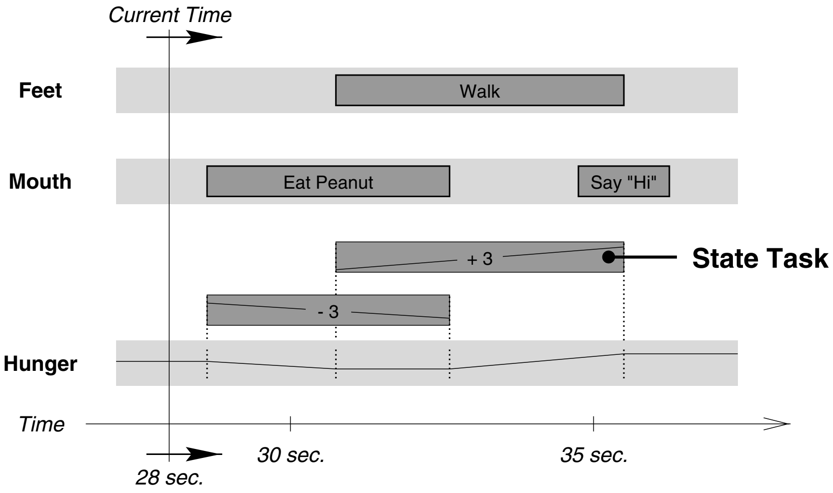


Fig. 4.3. The Mapping Mechanism of State Resources

There can be further effects, which may be caused by synergistic effects within a state resource. Adding water to a bathtub may result in its overflowing and wetting the bathroom. The actions cannot provide state tasks to realize these further effects because an action has only the limited view of its state task contributions. Thus, **dependency effects** of specific state resource states must be expressed in addition. The dependencies are special actions that are beyond the agent's control. Expected external events can also be integrated by these dependencies.

4.1.3 Objects and References

In a finite and known world, there is a fixed set of resources to be considered. Static relations to specific state resources can be used to realize the effects of actions, e.g., that the state tasks of the action Eat Peanut-17 affect the state resource Peanut-17 Location. But such static relations are no longer possible in an open world, where it is unclear which and how many resources exist[2]. Thus, as a consequence of the open world assumption, an action's tasks must be specified with variable **references** for the state resources involved (e.g., Eat Peanut X instead of Eat Peanut-17).

The next problem that arises by dropping the closed-world assumption is that the relations between the state resources themselves are no longer fixed. For example, there could be two peanuts, a big and a small one, and thus multiple state resources Peanut Location and Peanut Nutritive Value. The resources of the same type are indistinguishable, and it is not clear which two belong to a specific peanut. If the Eat Peanut action were applied, it is not clear which resources would be affected, and the big peanut might vanish, while the small one would be used to decrease hunger. Thus, in addition to an action's tasks, the state resources' states may involve references to indicate their relation.

The most common relation between state resources is an aggregation – they form **objects**. As this is a very important relation, it is not handled by references, but explicitly represented in the model. For example, the state resources Peanut Location and Peanut Nutritive Value form an object Peanut. Figure 4.4 illustrates the application of the Eat Peanut action to a Peanut object.

4.1.4 Sensors and Existence

The agent must be capable of acquiring new information about the environment, which must somehow be integrated into the planning process. The real-world data is collected by so-called **sensors**. The sensors report actual data, like the current level of hunger or the properties of a sighted peanut. We assume high-level sensoring that provides ready-structured objects. Sensors are related to the virtual objects of the plan. Figure 4.5 shows an example of a plan to put one block on top of another where only one of the blocks has already be sensed.

Having introduced the concept of sensors, we are faced with the question of whether a planning object will ever be connected to a sensor, i.e., if a counterpart in the real world actually exists or will exist. For example, it is pointless optimistically creating/revising a plan so that a matching Key is always found next to a locked Door. However, the existence of objects is not

[2] We assume that all possible types of resources are known, but not the number of instances.

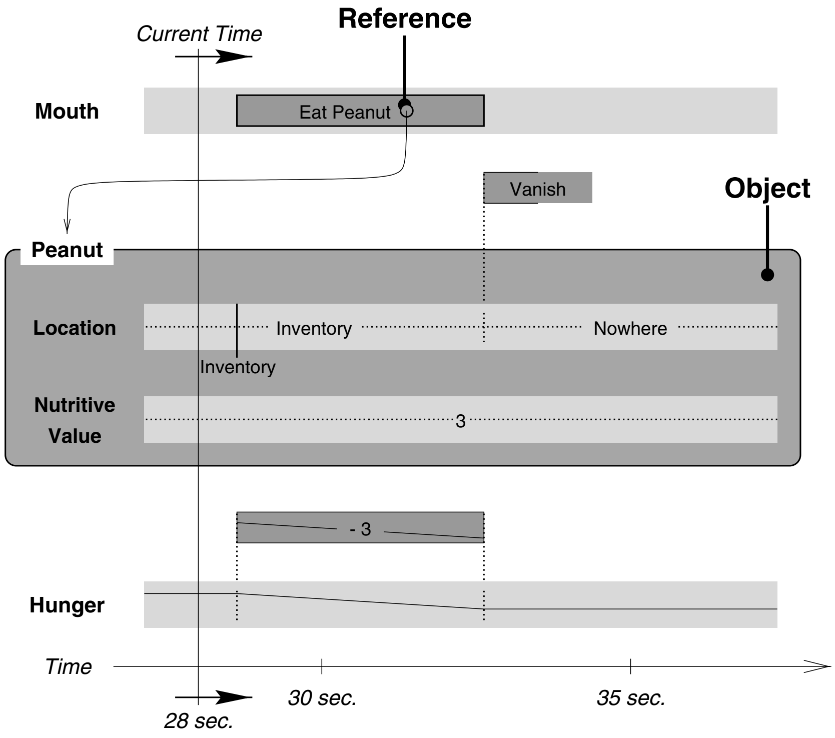


Fig. 4.4. References and Objects

a yes/no matter. It is temporally dependent (a KEY may become available *after* a LOCK is installed) and a probabilistic matter (the KEY *may* become available). Thus, we need a temporally projected probabilistic measure for every object, which expresses the confidence that this object really exists – an **existence projection**.

4.2 The Planning Model as SCSP

This section defines the planning model in terms of structural constraint satisfaction. The representation of the planning model as an SCSP allows us to apply the previous chapter's techniques to generate the structural search space. Figure 4.6 gives an overview of possible relations between the planning SCSP's elements.

4.2.1 The Current Time

The very first thing we need is a variable for the current time because the constraints' heuristics and cost/goal functions use this as input. For example,

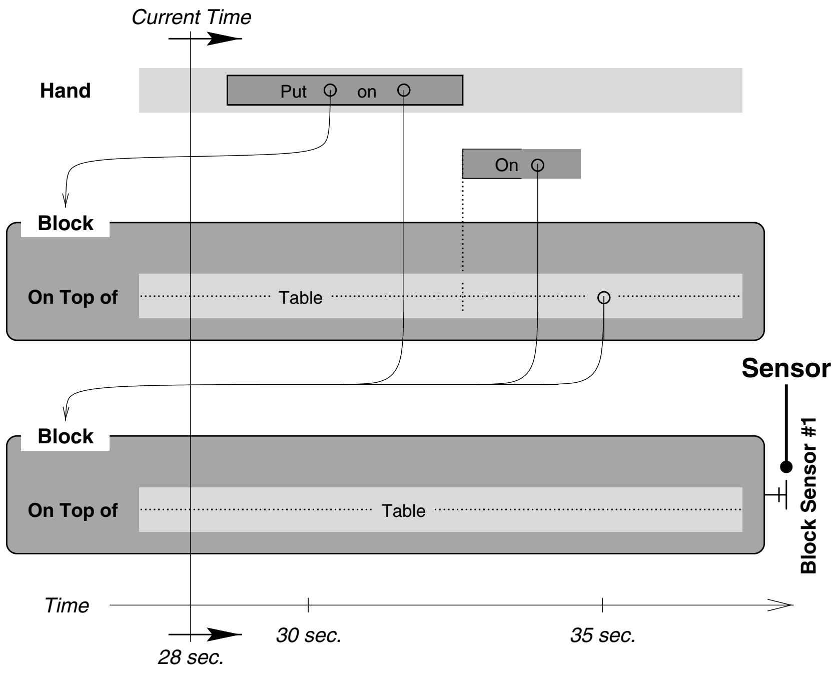


Fig. 4.5. Sensors

actions that are still to be executed should not be placed in the past. The variable is marked by a CURRENT TIME object constraint (see Fig. 4.7). As there can only be one current time, we need a structural constraint to prevent there being multiple variables representing the current time (see Fig. 4.8).

4.2.2 Actions

An action consists of a set of different preconditions, operations and resulting state changes. These elements are represented by tasks, i.e., there are PRECONDITION TASKS for precondition tests, ACTION TASKS for operations and STATE TASKS for state changes. All tasks are represented by object constraints and must be connected to a TASK CONSTRAINT.

A TASK CONSTRAINT enforces a certain task configuration (i.e., it specifies which tasks are to be connected to the TASK CONSTRAINT and what kind of restrictions apply to the tasks' variables) for a specific action, including the temporal order of the tasks. The specific action to be expressed is determined by the value of a connected *ActionType* variable (see Fig. 4.9). The cost function of the TASK CONSTRAINT describes the distance from the current task configuration to the configuration that is demanded by the *ActionType* variable. In addition, the *Begin* of nonexecuted ACTION TASKS before the CURRENT TIME is penalized.

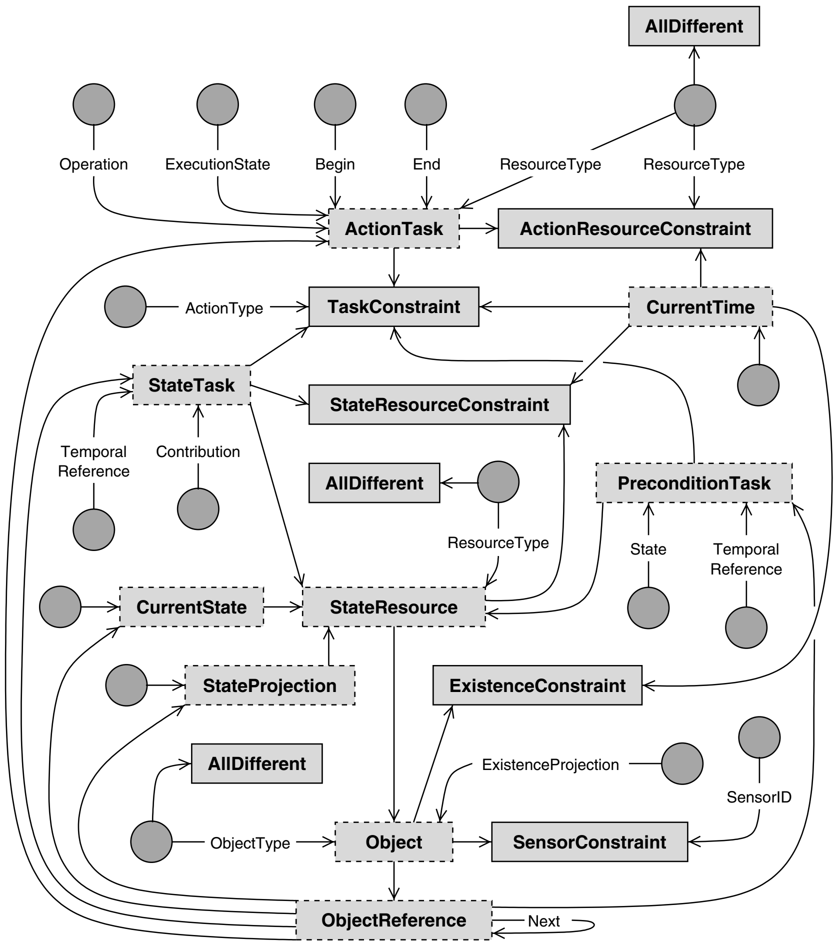


Fig. 4.6. Possible Type Relations

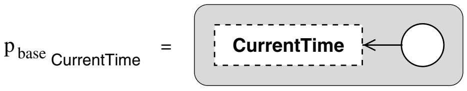


Fig. 4.7. The Extensible Object Constraint CURRENT TIME

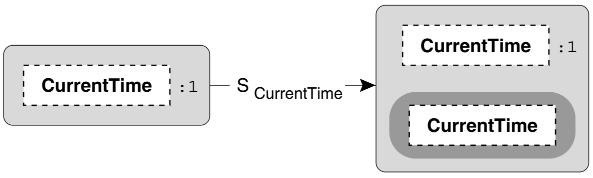


Fig. 4.8. The Structural Constraint CURRENT TIME

$p_{\text{base}\ \text{TaskConstraint}}$ =

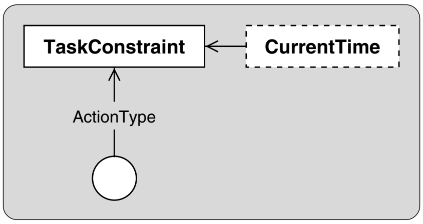


Fig. 4.9. The Extensible Conventional Task Constraint

An action's task may not also be part of another action. This is ensured by the structural constraint in Fig. 4.10.

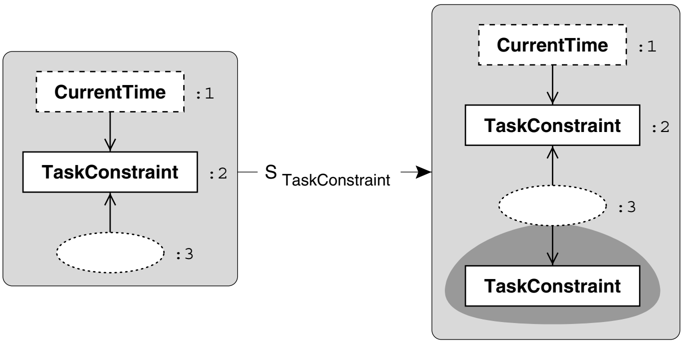


Fig. 4.10. The Structural Task Constraint

4.2.3 Operations

An Action Task specifies a concrete operation that must be executed within an action (see Fig. 4.11).

$p_{\text{base}\ \text{ActionTask}}$ =

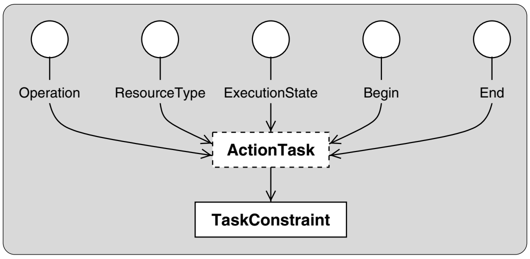


Fig. 4.11. The Extensible Object Constraint Action Task

An Action Task's operation uses a specific action resource. For an Action Task's duration, other tasks are required to leave enough of the action resource's capacity to carry out the task's operation. An Action Resource Constraint (ARC) internally projects an action resource's capacity and reflects an overload of the resource by its cost function (see Fig. 4.12).

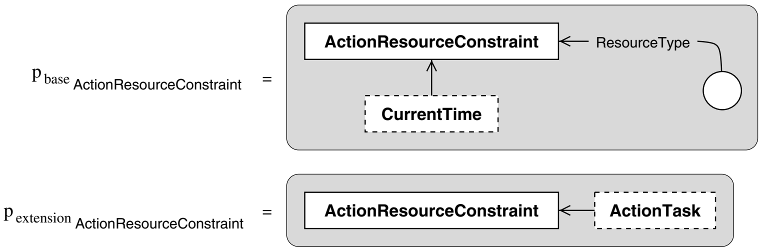


Fig. 4.12. The Extensible Conventional Action Resource Constraint

Each Action Task must be linked to a specific ARC of the required *ResourceType* (see Fig. 4.13). The Action Task that is to be executed at the Current Time on an action resource is determined by the ARC by demanding a special value for the task's *ExecutionState* variable.

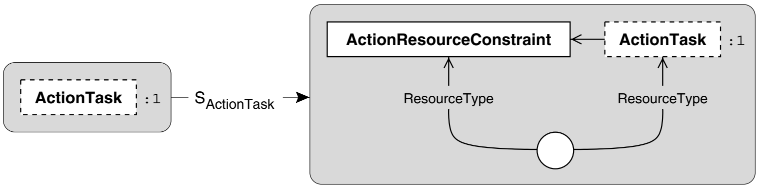


Fig. 4.13. The Structural Constraint Action Task

In addition, all ARCs must have a different *ResourceType*. To realize this, we can use an All Different constraint (see Fig. 4.14). This constraint ensures that all connected variables have different values. An ARC must have its *ResourceType* variable connected to an All Different constraint (see Fig. 4.15). The requirement that there be no other All Different constraint for different ARCs is enforced by the structural constraint of Fig. 4.30 (upper alternative of the testing part).

The uniqueness of an ARC's *ResourceType* also ensures that an Action Task cannot be connected to two ARCs at the same time.

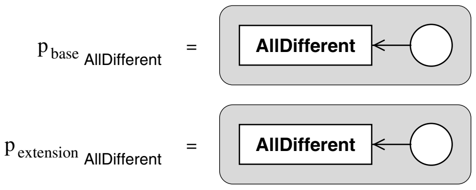


Fig. 4.14. The Extensible Conventional Constraint All Different

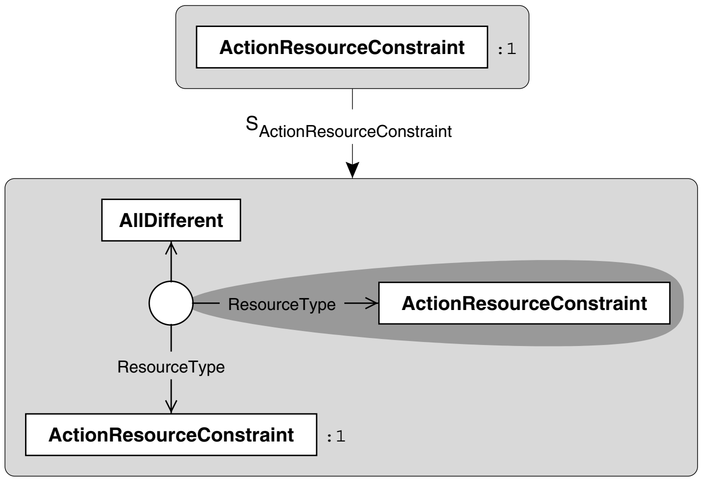


Fig. 4.15. The Structural Action Resource Constraint

4.2.4 States

Besides Action Tasks, an action consists of Precondition Tasks (see Fig. 4.16) and State Tasks (see Fig. 4.17). The reason why State Tasks do not have to be linked to a Task Constraint will be explained later.

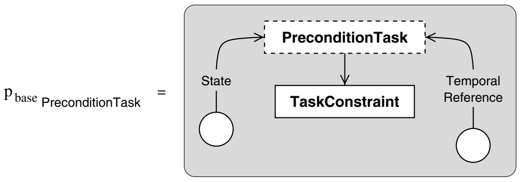
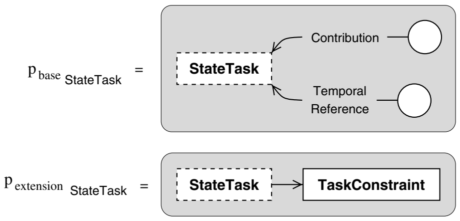


Fig. 4.16. The Extensible Object Constraint Precondition Task

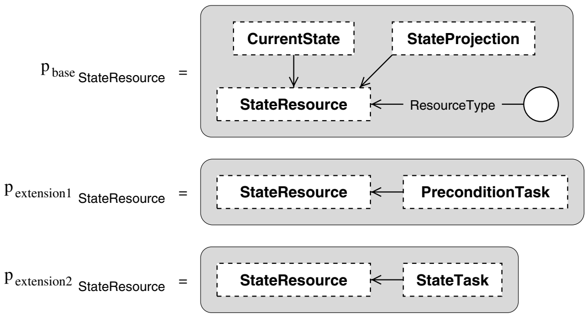
Unlike an action resource's structures, which are only internally represented in an ARC, a state resource's structures are explicitly stored in the model. This is because other constraints must access the state information as well (e.g., the Existence Constraint; see Sect. 4.2.8). Thus, Precondition



$p_{\text{base StateTask}}$ =
StateTask
Contribution
Temporal Reference
$p_{\text{extension StateTask}}$ =
StateTask
TaskConstraint

Fig. 4.17. The Extensible Object Constraint State Task

Tasks and State Tasks are linked to a State Resource object constraint which specifies the property that is to be tested/changed. A State Resource object constraint relates Precondition Tasks and State Tasks to a *ResourceType* variable, a State Projection and a Current State (see Fig. 4.18).



$p_{\text{base StateResource}}$ =
CurrentState
StateProjection
StateResource
ResourceType
$p_{\text{extension1 StateResource}}$ =
StateResource
PreconditionTask
$p_{\text{extension2 StateResource}}$ =
StateResource
StateTask

Fig. 4.18. The Extensible Object Constraint State Resource

All Precondition Tasks and State Tasks are required to be connected to exactly one State Resource (see Fig. 4.19 and 4.20).

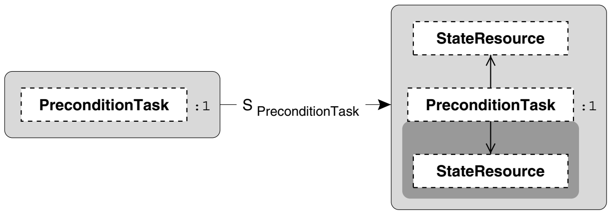


Fig. 4.19. The Structural Constraint Precondition Task

The Current State references a variable (see Fig. 4.21) that contains the State Resource's state at the Current Time. The State Projec-

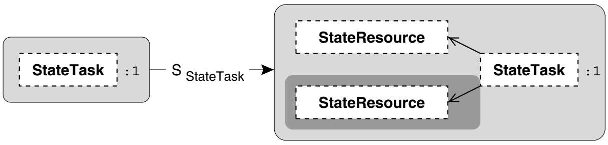


Fig. 4.20. The Structural Constraint State Task

tion references a variable (see Fig. 4.22) that stores the temporal projection of the resource's state for the whole timeline.

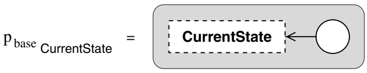


Fig. 4.21. The Extensible Object Constraint Current State

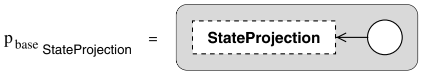


Fig. 4.22. The Extensible Object Constraint State Projection

Both, the Current State and the State Projection, must be connected to exactly one State Resource (see Fig. 4.23 and 4.24).

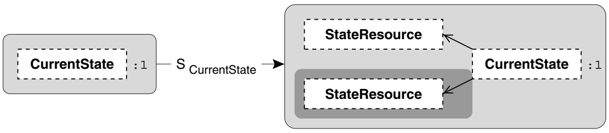


Fig. 4.23. The Structural Constraint Current State

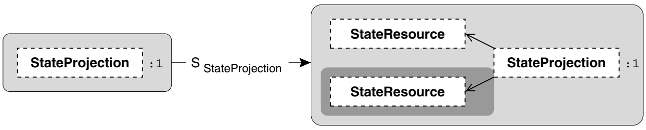


Fig. 4.24. The Structural Constraint State Projection

A State Resource Constraint (SRC) is linked to the State Resource to ensure a correct State Projection (see Fig. 4.25). The SRC uses the State Resource's Current State and the *Contributions* of the STATE RESOURCE's STATE TASKs to project the property's state development according to the STATE TASKs' *TemporalReferences* on a timeline, which is stored in the STATE PROJECTION's variable. The constraint's costs are computed according to satisfaction of the assigned precondition tests.

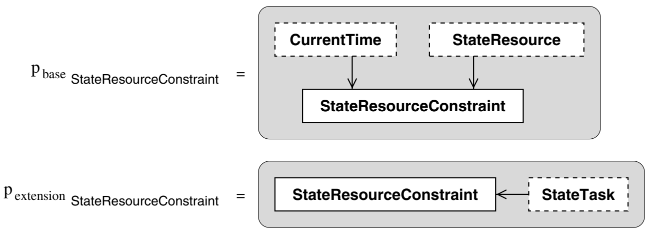


Fig. 4.25. The Extensible Conventional STATE RESOURCE CONSTRAINT

The structural constraint of Fig. 4.26 ensures that each STATE RESOURCE is connected to exactly one OBJECT and that an SRC is connected. Although it is not necessary that all *ResourceType* variables of the STATE RESOURCEs have different values, this facilitates searching, e.g., for all NUTRITIVE VALUE STATE RESOURCEs of the current plan. For this purpose, all *ResourceType* variables must be connected to an ALL DIFFERENT constraint.

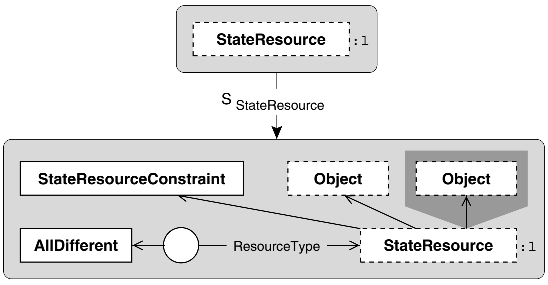


Fig. 4.26. The Structural Constraint STATE RESOURCE

The SRC is also responsible for maintaining the dependency effects of the resource's state development. To accomplish this, STATE TASKs can be connected to an SRC. Thus, a STATE TASK must either be connected to exactly one STATE RESOURCE CONSTRAINT or one TASK CONSTRAINT. This is ensured by the structural constraint of Fig. 4.27 (together with that of Fig. 4.10). A configuration of an SRC's STATE TASKs that does not represent the correct dependency effects has an impact on the value of the SRC's cost function.

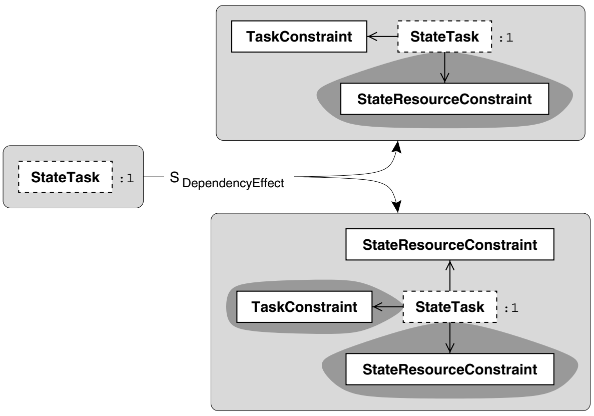


Fig. 4.27. The Structural Constraint Dependency Effect

4.2.5 Objects

The Object aggregation is shown in Fig. 4.28. The role of the *Existence-Projection* variable and the Existence Constraint are explained in Sect. 4.2.8.

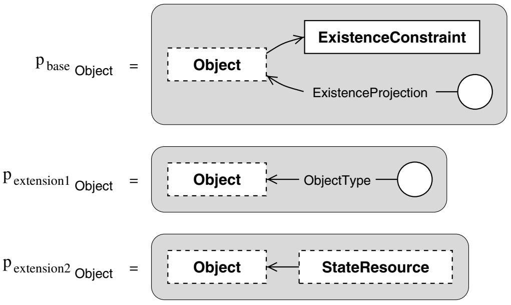


Fig. 4.28. The Extensible Object Constraint Object

The *ObjectType* variable specifies the type of the Object – for example, a Door. The *ObjectType* variable is not directly included in the p_{base} graph, as this allows the search (using the production generation of Chap. 3) to exchange an Object's *ObjectType* for a similar one without deleting the whole Object, e.g., if the search decides to consider an Apple instead of a Pear. However, this necessitates the structural constraint of Fig. 4.29, which ensures that an Object has exactly one *ObjectType* variable. In addition – as with the SRC – all *ObjectType* variables are connected to an All Different constraint.

Instead of a general unique identifier, the *ResourceType* of a State Resource has the task of specifying the State Resource's role within the

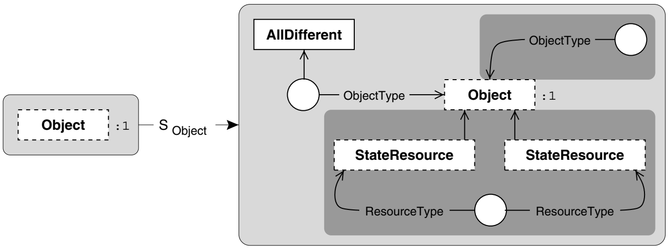


Fig. 4.29. The Structural Constraint OBJECT

OBJECT, e.g., a DOOR's LOCATION, LOCK or COLOR. To prevent ambiguities, all *ResourceType* variables of an OBJECT's STATE RESOURCE must be different (see Fig. 4.29).

For an ALL DIFFERENT constraint, an additional structural constraint is needed that enforces that the ALL DIFFERENT constraint must either be connected to an ARC's or STATE RESOURCE's *ResourceType* variable or an OBJECT's *ObjectType* variable, and that there must be no other ALL DIFFERENT constraint for the same kind of type variables (see Fig. 4.30).

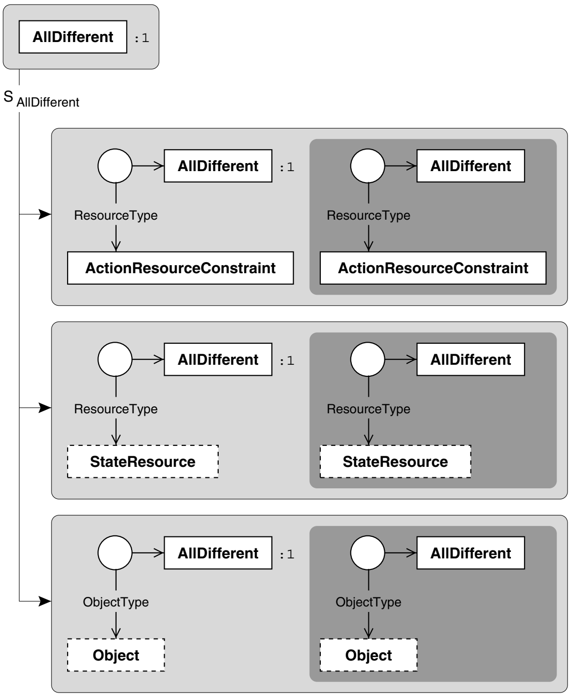


Fig. 4.30. The Structural Constraint ALL DIFFERENT

Any State Resource must be linked to an Object (see Fig. 4.26). Agent-internal State Resources, e.g., the agent's Hunger, can be linked to a unique Ego Object.

A Task Constraint (or SRC in the case of dependency effects) must also ensure that the connection of *ResourceTypes* and *ObjectTypes* with the Task Constraint's (SRC's) State Tasks and Precondition Tasks is correct with respect to the Task Constraint's *ActionType* (*ResourceType* of the SRC's State Resource). For example, an action Eat Peanut X should link its State Task with a *Vanish* contribution to the same Object that its Precondition Task with the location test is linked to.

4.2.6 References

All tasks can use Object References to Objects, e.g., a general State Task that causes one Object X to be on top of another Object Y can make use of Object References to assign its variables X and Y. Likewise, the Current State and the State Projection can have Object References (see Fig. 4.31). An Object Reference can also be linked to another Object Reference to realize a list. For example, the State Task of the previous example would have to use a list (instead of a set) with two Object References to distinguish the two Objects (which Object has to be placed onto which). The Task Constraint (or SRC in the case of dependency effects) must ensure that the reference structures are valid.

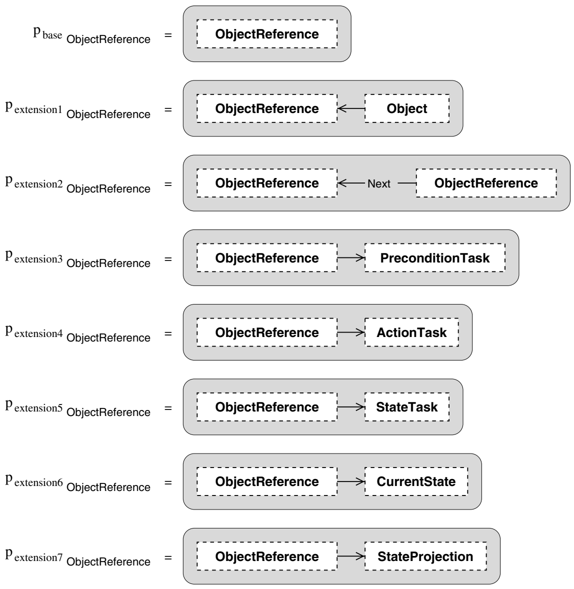

p_{base} ObjectReference =
ObjectReference
$p_{extension1}$ ObjectReference =
ObjectReference
Object
$p_{extension2}$ ObjectReference =
ObjectReference
Next
ObjectReference
$p_{extension3}$ ObjectReference =
ObjectReference
PreconditionTask
$p_{extension4}$ ObjectReference =
ObjectReference
ActionTask
$p_{extension5}$ ObjectReference =
ObjectReference
StateTask
$p_{extension6}$ ObjectReference =
ObjectReference
CurrentState
$p_{extension7}$ ObjectReference =
ObjectReference
StateProjection

Fig. 4.31. The Extensible Object Constraint Object Reference

All Object References are required to be connected to exactly one Object that is to be referenced (see Fig. 4.32).

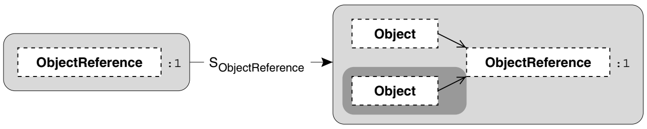


Fig. 4.32. The Structural Constraint Object Reference

4.2.7 Sensors

The sensors provide data that is structured according to the planning model's Objects. The linking of a sensor with an appropriate Object is done by a Sensor Constraint that links a *SensorID* variable, which specifies the real-world's sensor, with an Object (see Fig. 4.33). The Sensor Constraint ensures that the Object's *ObjectType* variable corresponds to the sensor data and that the Object has all necessary State Resources. The productions for an addition/deletion of a Sensor Constraint are not allowed to be applied by any improvement heuristic. If the Sensor Constraint is satisfied, the sensor data can provide values for the *CurrentState* variables of the connected Objects' SRCs.

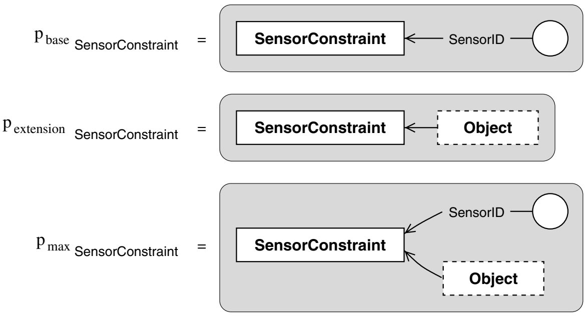


Fig. 4.33. The Extensible Conventional Sensor Constraint

Again – as with an Object's *ObjectType* – a Sensor Constraint's Object is not directly included in the p_{base} graph, as this allows the search to exchange a Sensor Constraint's Object for a similar one without deleting the whole Sensor Constraint, e.g., if the search decides that a sensed Peanut is a new one instead of the Peanut that has been eaten. However, this necessitates the structural constraint of Fig. 4.34, which ensures that a sensor is linked to an Object.

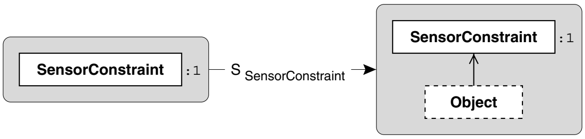


Fig. 4.34. The Structural SENSOR CONSTRAINT

4.2.8 Existence Projections

The confidence projection that an OBJECT really exists is realized by using an *ExistenceProjection* variable, which is similar to the temporal *StateProjection* variables of SRCs, taking values between 0 and 1 at each time point (see Fig. 4.28).

The *ExistenceProjection* is not only dependent on OBJECT-local features but on the whole world state. For example, it is improbable that there is a DOOR if the agent's LOCATION is PARK. Given these global conditions, an EXISTENCE CONSTRAINT must be linked to all OBJECTS to be capable of accessing all relevant STATE RESOURCES. To keep the linking costs reasonably low, only one EXISTENCE CONSTRAINT is responsible for maintaining all OBJECTS' *ExistenceProjections.*

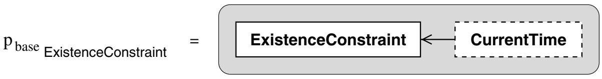


Fig. 4.35. The Extensible Conventional EXISTENCE CONSTRAINT

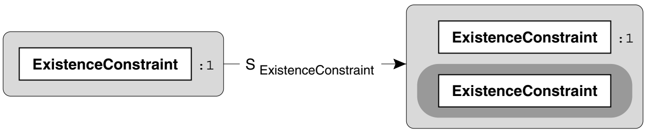


Fig. 4.36. The Structural EXISTENCE CONSTRAINT

The EXISTENCE CONSTRAINT must also ensure correct OBJECT configurations. For example, an OBJECT with an *ObjectType* of DOOR and a NUTRITIVE VALUE STATE RESOURCE gets a very poor *ExistenceProjection.* However, OBJECTS do not have to be specified completely. For example, the DOOR's COLOR STATE RESOURCE may not be important for a plan to open it.

The *ExistenceProjection* of an OBJECT has an impact on the satisfaction of TASK CONSTRAINTS that have tasks connected to the OBJECT. Using this mechanism, an appropriate temporally projected anchoring, i.e., a convenient correspondence between the plan's and the real world's objects (see [33] for a more detailed treatment of this topic), can be realized.

4.2.9 Correctness

To apply the SCSP handling methods described in Sect. 3.4, some restrictions must be fulfilled.

– Nonextensible constraints are not allowed to appear in graphs of other constraints, as the addition and deletion productions of constraints partly incorporate their graphs.

The specification of the planning SCSP does not include any nonextensible constraints.

– Constraint-usage cycles are not allowed for the p_{base} graphs of extensible constraints, e.g., that p_{base_A} includes a B constraint and p_{base_B} includes the A constraint.

Figure 4.37 confirms that there are no cycles for the planning problem's specification.

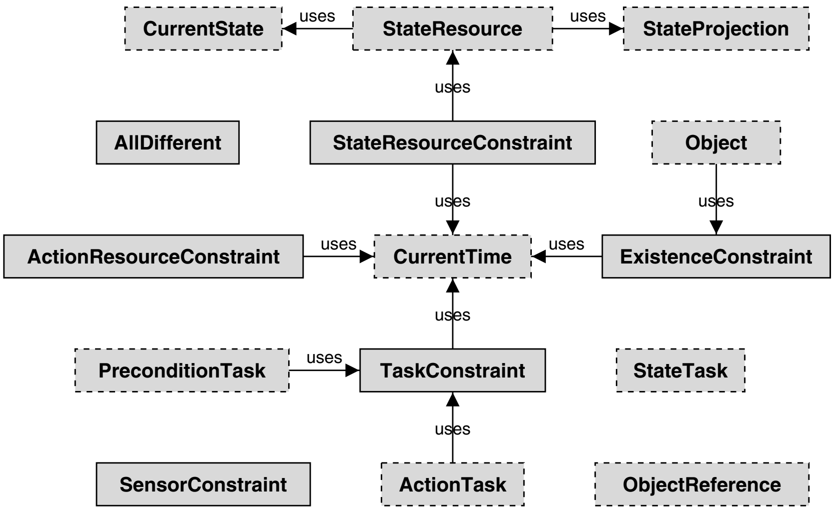


Fig. 4.37. Integrations of Extensible Constraints' Base Graphs

– If the p_{max} graph of an extensible constraint is non-empty, no sequence of productions that is applied to the constraint's p_{base} graph can produce a graph that includes the p_{max} graph without previously producing the p_{max} graph.

The only constraint with a p_{max} graph is the SENSOR CONSTRAINT. The only way to extend the $p_{base_{SensorConstraint}}$ graph is by using the $p_{extension_{SensorConstraint}}$ graph, which immediately produces the $p_{max_{SensorConstraint}}$ graph.

4.2.10 Problem Formulation

The SCSP of the previous sections specified a general planning problem. However, for a specific problem, additional domain information must be provided. The constraints must be able to handle the specific domain values and must have appropriate cost/goal functions and improvement heuristics, e.g., the TASK CONSTRAINT must know about the permitted action configurations and the EXISTENCE CONSTRAINT must be able to project the existence confidence for the domain's OBJECTS.

A specific planning problem can include satisfaction goals (e.g., the door *is to be open* at time point 2507) and optimization goals (e.g., the minimal hunger level over time *is to be as high as possible*). Satisfaction goals can be represented by TASK CONSTRAINTS that have only PRECONDITION TASKS, and optimization goals can be realized by initialization of the resource constraints' goal functions[3].

The satisfaction goals and optimization goals must be embedded in the search process. Thus, we can define additional constraints that restrict variables to a specific value (a constant) and require that these constraints exist in a solution where they, for example, restrict a TASK CONSTRAINT's *ActionType* variable and the STATE RESOURCES' *ResourceType* variables (e.g., see Fig. 4.38).

4.3 Incomplete Knowledge

An agent's incomplete knowledge of his environment complicates the planning process enormously. The model of the previous sections has already addressed incomplete knowledge with respect to the existence of entities. This section deals with incomplete knowledge regarding the values of certain properties. In contrast to the existence matter, uncertain property values are handled by special value assignments for the state resources.

4.3.1 A Single-Plan Approach

If a decision relies on an unknown property, every possible property value may yield another plan. Planners that construct branching plans for unknown properties are called *contingency planners*. Examples are Warren's

[3] Of course, the whole expressiveness of the SCSP approach can be used to formulate much more complicated goals, but in most cases a representation using specific TASK CONSTRAINTS / resources' goal functions is adequate to model the goals.

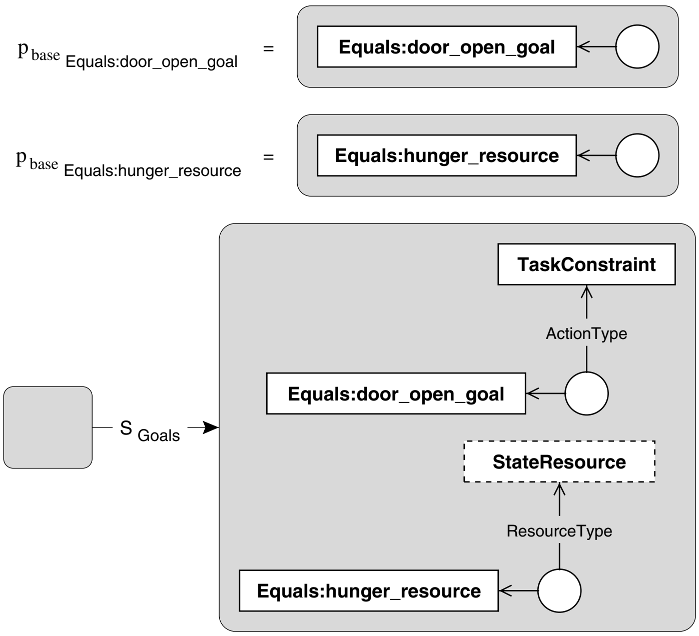


Fig. 4.38. Specification of the Goals' Existence

WARPLAN-C [187], CNLP by Peot and Smith [148], Plinth by Goldman and Boddy [71] and Cassandra by Pryor and Collins [151].

An extension of this approach is *probabilistic planning*, where special probability distributions are considered as well. Work in this area includes Drummond and Bresina's synthetic projection [48] and the BURIDAN probabilistic planning by Kushmerick, Hanks and Weld [108] with its contingent extension by Draper, Hanks and Weld [47]. Recently, much research in this area has focused on planning based on Markov decision processes (MDPs) (see [22] for an overview). Possible world states are represented explicitly, an optimal policy being computed for them. This policy yields the optimal action for every possible state. Examples of MDP-based planning are approaches by Barto, Bradtke and Singh [14], Dean et al. [37] and Koenig and Liu [104].

The consideration of/branching on every possible value of an unknown property can be useful in terms of reliability. But this works only for small-scale problems. Although planners (especially the probabilistic ones) do not always search the whole search space, their application to more complex problem domains, temporal planning and dynamic environments would greatly overtax memory and computation capacity. Strong real-time requirements such as those for our agents are out of the question.

One solution is to consider only *one plan* with expected values instead of branching on several possible worlds. The expectations can be based on pessimistic or optimistic estimates, as well as on estimated probabilities. Learning mechanisms can be applied, too. In the case of a failed prediction, the plan has to be changed. Provided no critical errors have been made, the iterative plan repair can automatically adapt the plan. The consideration of a single plan possibility instead of branching on multiple possibilities is similar to the operator-parameterization approach of the Cypress system [189]. A non-branching plan can also represent incomplete knowledge regarding the states by using special values that represent the incomplete knowledge (see following sections).

4.3.2 Missing Information

The state resources can represent the lack of information by the addition of an *Unknown* value to their domain (which gets the default state). For example, in the agent's absence, other agents might open or close a door without the agent's noticing. Whether the door is closed or not can only be known if the door is within the agent's field of vision. Thus, the state resource Door with a domain of *Open* and *Closed* gets an additional *Unknown* value, which is triggered by actions that cause the agent's absence.

An action of passing the door requires that the Door is open. In a pessimistic approach, the precondition task of the passing action entails a bad satisfaction of the state resource if the Door is in an *Unknown* state because failure could endanger later commitments. The inclusion of a prophylactic Open Door action would avert this threat (see Fig. 4.39).

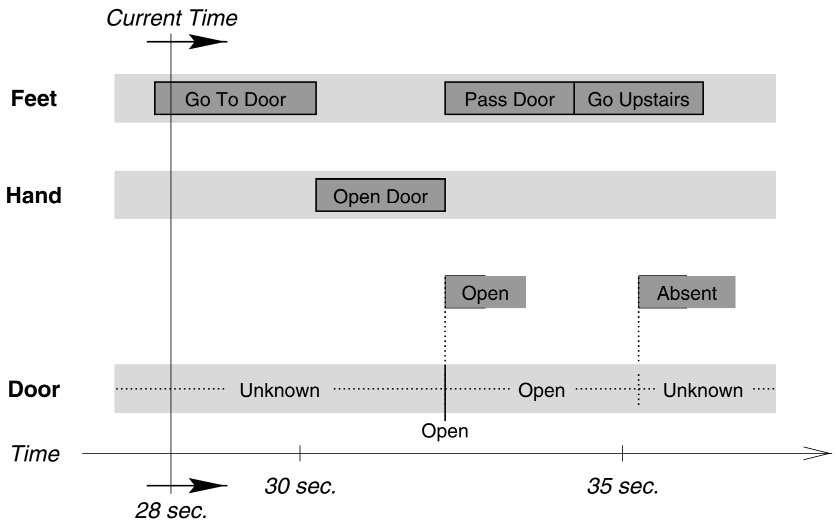


Fig. 4.39. The Use of *Unknown* Values

On the other hand, the satisfaction could be driven by experience. If the agent has learned that the Door is usually open, the state resource Door might settle for the *Unknown* state with respect to a precondition check if the door is open.

4.3.3 Information Gathering

Classical planners normally try to satisfy all goal states. But in an incomplete environment they cannot decide whether an unknown state is already satisfied or not. An unknown state like COLOR(DOOR, BLUE) could only be satisfied by painting the door blue. If the door is already blue, this action would be unnecessary, and a lack of blue paint would even entail an inconsistency. The ability to plan sensory actions too was realized in various STRIPS-based approaches, such as IPEM by Ambros-Ingerson and Steel [9], UWL and XII by Etzioni et al. [54] [70], Sage by Knoblock [107] and Occam by Kwok and Weld [109].

Run-Time Variables. The concepts are mostly based on run-time variables, which are initialized by sensing actions. In our model, the fact of knowing a state can easily be expressed by the inclusion of *Known* values for state resources. These values are triggered by actions, which include the sensing of the state resource's property. The state resources' mapping mechanisms must protect already specified states from the more general *Known* reassignment, and allow a switch to the *Known* state only from *Unknown* states. This process does not differ from the normal state processing and does not require a special sensory treatment. In the example of Fig. 4.40, the crossing of a bridge in an *Unknown* state is considered to be unsatisfiable enough to include an additional TEST BRIDGE action.

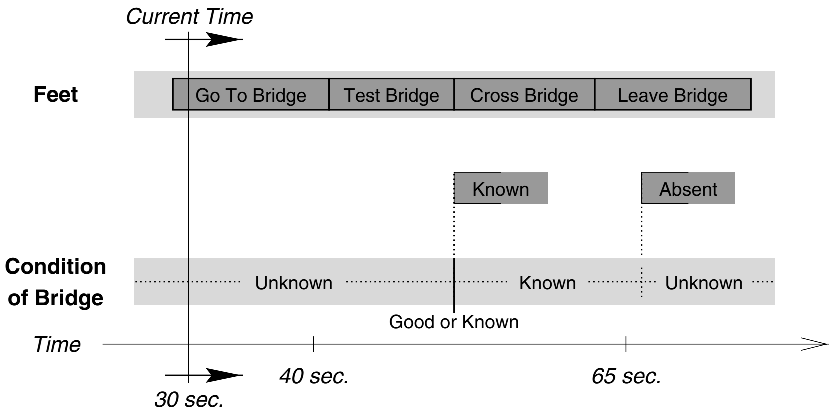


Fig. 4.40. The Use of *Known* Values

The problem of *redundant sensing*, which is addressed in [70] is not present here, as there is only a single plan and the state resources' information is accessible over the whole time period.

Hands-Off Goals. *Hands-off goals* [54, 70] that forbid the change of states are not necessary either. The formulation of goals like passing the blue door is a problem only for planners, who cannot express that the door has to be blue at the moment of the goal formulation. These planners have to introduce such hands-off goals to prevent the planner from passing another door after painting it. Temporal planners can express these goals much more adequately because they can quantify the preconditions temporally (see Fig. 4.41). Moreover, in dynamic multi-agent domains, such hands-off goals do not help, as external actions might change the states.

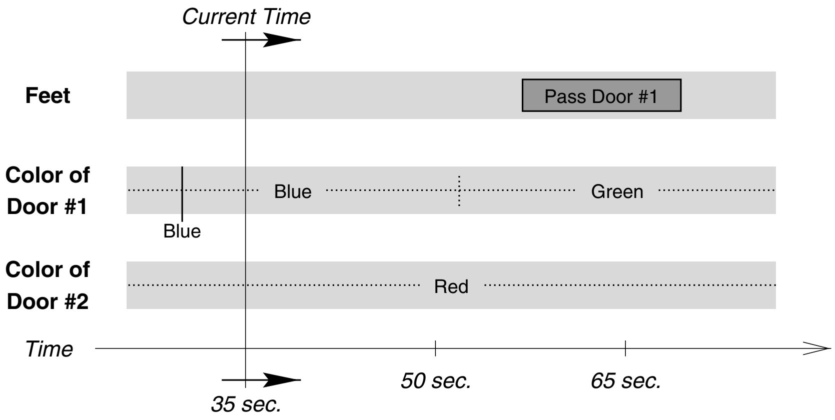


Fig. 4.41. Temporal Quantification

4.3.4 Partial Knowledge

So far, only state resources with two-valued domains and clear transitions between knowledge and no knowledge have been considered. The following sections refine this approach.

Unordered Domains. State resource domains in our agent model may not only be two-valued, like the DOOR with *Open* and *Closed* values. For example, there may be an additional *Locked* value. In a situation of incomplete knowledge, each value must have an associated knowledge level of

– *Unknown*: The information as to whether the value corresponds to the state is not available.
– *Known*: The information as to whether the value corresponds to the state will be available in the future.
– *Not*: The domain value definitely does not correspond to the state.

Figure 4.42 shows an example of an agent going away and leaving a door open for which the agent has the only key.

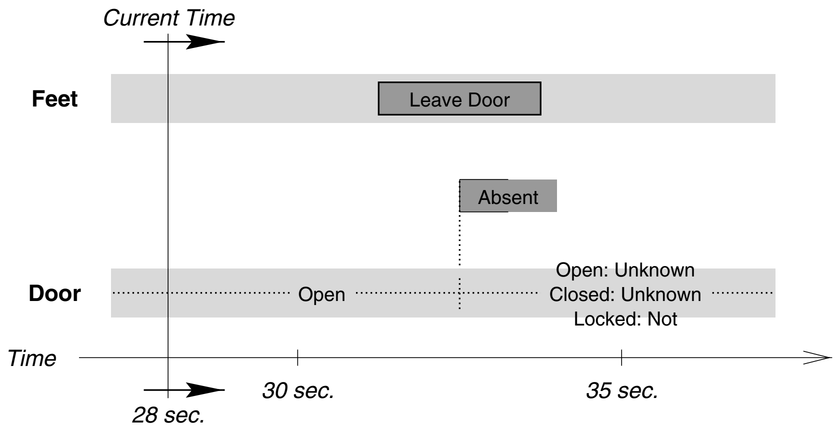


Fig. 4.42. Incomplete Knowledge of State Values

Because of the XOR-relation of the domain values, some propagations can be made within a state resource. If only one domain value has a knowledge level of *Unknown*, then this value gets the knowledge level *Known*. If only one domain value is not *Not*, then this value must be the state resource's state.

Ordered Domains. In contrast to the value sets of the previous section, the elements of a state resource's domain can also be in a specific ordering relation. For example, the agent wants to fill a bucket with water, but he does not know how much water is in the bucket to start with. The amount of water in the bucket can be modeled by an integer state resource.

Of course, it is possible to apply the *Unknown-Known-Not* representation from the previous section to each domain value. But this would be a rather costly approach. It is more efficient to subsume consecutive values of the same knowledge level by intervals[4]. Figure 4.43 shows an example.

It is even more efficient to consider only the convex (vertical) hull of intervals (trapezoids) of the same knowledge level. Actually, this method is not precise, because already excluded intermediate values may not be accessible any more. Consequently, the intervals (trapezoids) of different knowledge levels might overlap. For example, in the 65th second of Fig. 4.43, the convex hull of the *Not* knowledge level includes the *Known* values.

Probabilities. Probabilities for prospective states of the state resources are not a basic part of the model, because this is not generally needed and would waste a lot of system resources. In some cases, however, the use of probabilities can dramatically affect the plan result, especially if a state probability can be changed by special actions. For example, if the agent is searching for a key, it is not easy to express progress within the search process without

[4] An appropriate representation for taking into account the dimension of time are trapezoids (provided that changes are only linear).

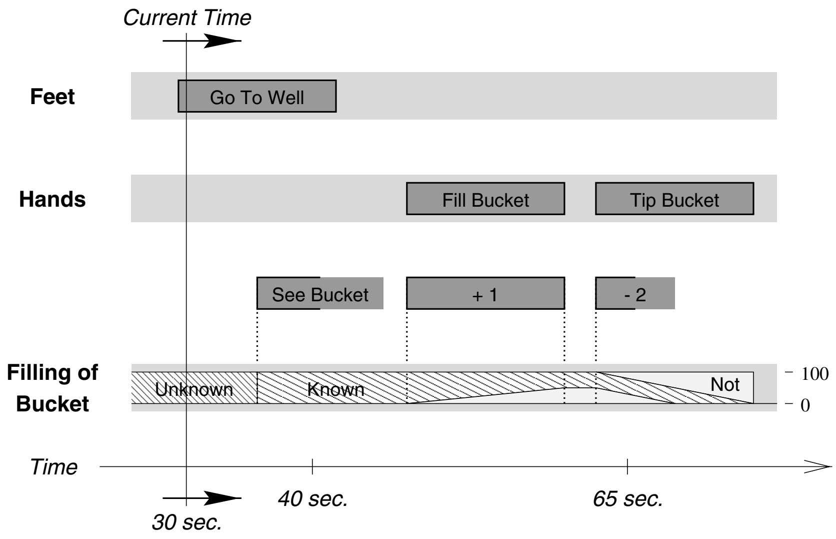


Fig. 4.43. Value Subsumption

probabilities. Whether the agent searches one drawer or two drawers must make a difference.

The probabilities can be realized by continuous domain state resources. Figure 4.44 shows a possible modeling of the search for the key.

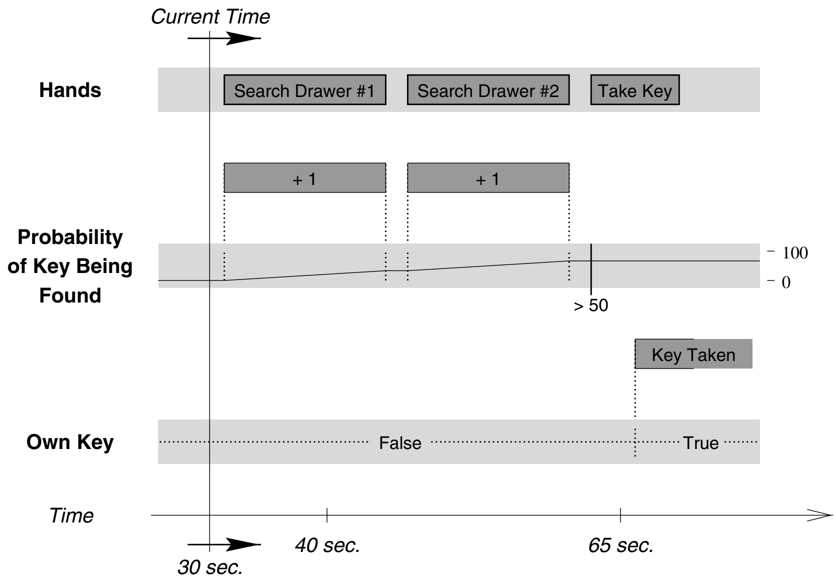


Fig. 4.44. State Probabilities

4.4 Conclusion

For planning problems that involve resource constraints, some planning systems keep planning and resource allocation separate, e.g., the approach of Srivastava and Kambhampati [169] and the *parc*PLAN system [50]. The separation of planning and resource allocation prevents the systems from considering interactions between the decisions with respect to planning and resource assignment, which is a great disadvantage if resource-related properties are to be optimized. Consequently, the resources serve only as constraints for the planning problem and are not used as primary optimization goals. The same applies to resource-based planning systems that integrate planning and resource allocation, like O-Plan2 [44], IxTeT [110], the LPSAT engine's application to planning [191] and IPP's extension [103]. All of these focus primarily on optimization of the plan length, which is a rather curious approach as this property is usually irrelevant[5].

Resource-based planning systems that are based entirely on general search frameworks like CP or OR require bounds for the plan length or the number of actions. If a correct plan cannot be found, these bounds can be expanded. Some examples here are the OR-based approach of Bockmayr and Dimopoulos [18], ILP-PLAN [101], CPlan [175] and the approach of Rintanen and Jungholt [157]. Again, these systems primarily optimize the plan-length property. An optimum with respect to resource-related optimization goals can only be found if the initial bound can be set such that the optimal solution is guaranteed to lie within this bound. This is a very hard task for specific problems and impossible at a general level. Besides, creating the maximal structures for the search space is much too costly for complex real-world problems.

The presented constraint-based planning model does not require any bounds for the plan length and allows us to integrate the planning and scheduling process and focus on resource-related optimization. The model is based on the SCSP approach and avoids the use of maximal structures by including the search for the structure as part of the satisfaction process.

The SCSP approach also allows us to tackle open-world problems in which an arbitrary number of objects can be involved. The past decade has seen the development of a few planning systems with this capability, e.g., XII [70] and PSIPLAN [10]. Satisfaction-based planners are very rare in this domain. The lack of satisfaction-based approaches to open-world planning can be explained by the massive explosion of the search space under consideration. Systems based on maximal structures cannot handle this.

The presented model involves a representation of incomplete knowledge. In contrast to contingent/MDP planning approaches, the representation here does not include alternative timelines with branching points. Only one timeline is considered, which has to be adapted in case of a failed prediction (see

[5] See Sect. 1.3.2 for a discussion of decision-theoretic planning approaches.

Sect. 4.3.1). Probabilities and incomplete knowledge about states are expressed by special values, annotations and continuous domain state resources. A limited handling of an agent's partial view of the world state can thus be realized, avoiding the combinatorial explosion of branching approaches.

The presented planning model borrows from typical constraint-based applications for resource allocation/optimization. The power of global constraints for constraint-specific representation and reasoning mechanisms for specific resource types was recognized here very early on and led to significant speedups in the solution process. General frameworks for planning and scheduling like Muscettola's HSTS [124] lack such specialized representation and reasoning capabilities[6].

Furthermore, the model uses a domain-independent representation. By contrast, constraint-based planning systems like *parc*PLAN and CPlan – besides their inability to search in a way that is not focused on the criterion of plan length – do not have a model for general planning, relying instead on domain-dependent encodings.

[6] Work to overcome these deficiencies has, however, recently started [184].

5. Application

This chapter shows how the techniques developed in the previous chapters are combined to realize the planning of an agent's behavior. Section 5.1 presents a domain-dependent solution for the Orc Quest example described in Sect. 1.3.2. For a domain-independent system, more general constraint implementations must be available, which is detailed in Sect. 5.2.

The planning process follows the procedure shown in Fig. 5.1.

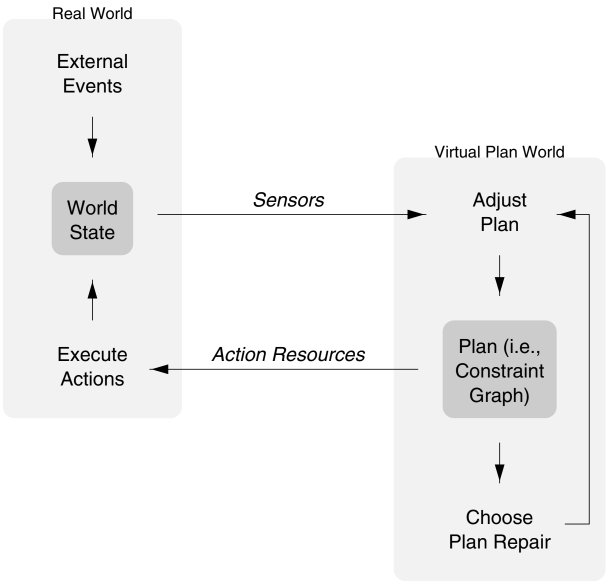


Fig. 5.1. Interleaving Sensing, Planning and Execution

In each iteration, the sensor data is used to update the current plan (i.e., the constraint graph). This means that for all sensed OBJECTs the CURRENT STATE variables of the OBJECTs' STATE RESOURCEs are updated. If a sensed real-world object has no counterpart in the constraint graph, a corresponding OBJECT is added to the graph. In addition, the CURRENT TIME variable is updated.

Once the plan has been updated, a plan repair is initiated[1]. A plan repair means that a constraint is selected by the global search control to improve the constraint's current goal-/cost-function value. The constraint's heuristics determine how the constraint will change some of its variables' values or/and use graph productions to change the constraint graph.

The planning iteration is ended by marking the action resources' ACTION TASKS that begin at the CURRENT TIME as being under execution (using their *ExecutionState* variable), by marking the ACTION TASKS that end at the CURRENT TIME as being executed, and by starting/continuing the execution of the ACTION TASKS that are marked for execution.

Usually, the structural constraints have to be enforced in the same way as the other constraints. However, to implement this is costly and not feasible within the context of this book. Instead, the enforcement of the structural constraints is done in a pre-emptive way: only heuristics may be used which can be shown not to threaten any structural constraints.

5.1 Revisiting the Orc Quest Example

Section 1.3.2 introduced the Orc Quest example, a solution of which is presented in this section. The problem was defined as:

```
STATEVARS: Duration, Pain, Performers ∈ ℕ0

INIT: Duration = 0, Pain = 0, Performers = 0

ACTION catch_only_one:
 Duration += 2, Pain += 1, Performers += 1

ACTION catch_a_group:
 Duration += 5, Pain += 4, Performers += 3

ACTION deliver_humans:
 Duration += 1, Pain -= 11, Performers -= 10
```

The planning goal is a multi-objective one. The goal criteria are:

```
GOAL:
 Satisfaction criterion: Performers ≥ 5
 Primary optimization criterion: min(Pain)
 Secondary optimization criterion: min(Duration)
```

5.1.1 The Constraints

Besides a general ORC ACTION RESOURCE CONSTRAINT, no other action resource constraints are needed as there is no further refinement regarding

[1] In most applications, there is some kind of time limit for a planning iteration (e.g., see [194]). In this case, multiple plan repairs can be executed in one planning iteration as long as the time limit is not exceeded.

the Orc's hands, feet, mouth or whatever. The Orc Action Resource Constraint can be linked to action tasks with a duration of 2 hours (if the task's *Operation* variable has the value `catch_only_one`), 5 hours (for a value of `catch_a_group`) and 1 hour (for a value of `deliver_humans`).

In addition, two state resource constraints are needed, the Pain State Resource Constraint and the Performers State Resource Constraint, which support discrete numerical domains and integrate state tasks by adding their pain/performers values. For example, the Pain State Resource Constraint's projection of the Orc's pain will be increased by 4 if a state task that has a *Contribution* variable with a value of 4 is connected.

Four different configurations satisfy a task constraint. These express the three actions `catch_only_one`, `catch_a_group` and `deliver_humans`, each of them having an action task for the Orc Action Resource Constraint, a state task for the Pain State Resource Constraint and a state task for the Performers State Resource Constraint. A fourth configuration consists of a precondition task for the Performers State Resource Constraint to establish the satisfaction goal `Performers` $\geq$ `5`. In addition, the task constraint requires that its tasks' variables have the appropriate values (e.g., if the task constraint's *ActionType* variable has a value of `catch_a_group`, a value of 4 is required for the *Contribution* variable of the state task that is to be connected to the Pain State Resource Constraint).

The Performers constraint's cost function is initialized to return the current plan's number of missing performers (to satisfy the precondition task of a task constraint with an *ActionType* variable that represents the task constraint's fourth configuration), the Pain constraint's goal function to return the plan's resulting pain, and the Orc constraint's goal function to return the plan's total duration. The specification of the problem's S_{Goals} constraint is shown in Fig. 5.2.

The problem does not inlude issues of sensing or uncertainty, i.e., existence constraints and sensor constraints must not be considered.

5.1.2 The Constraints' Heuristics

The problem does not require an application of sophisticated heuristics. The improvement heuristics can be the same for the Performers, Pain and Orc constraint. To change the plan (i.e., the CSP), there are six modification alternatives[2]:

- add a `catch_only_one` task constraint
- delete a `catch_only_one` task constraint
- add a `catch_a_group` task constraint

[2] The pre-emptive maintenence of the structural constraints in explained in Appendix D.

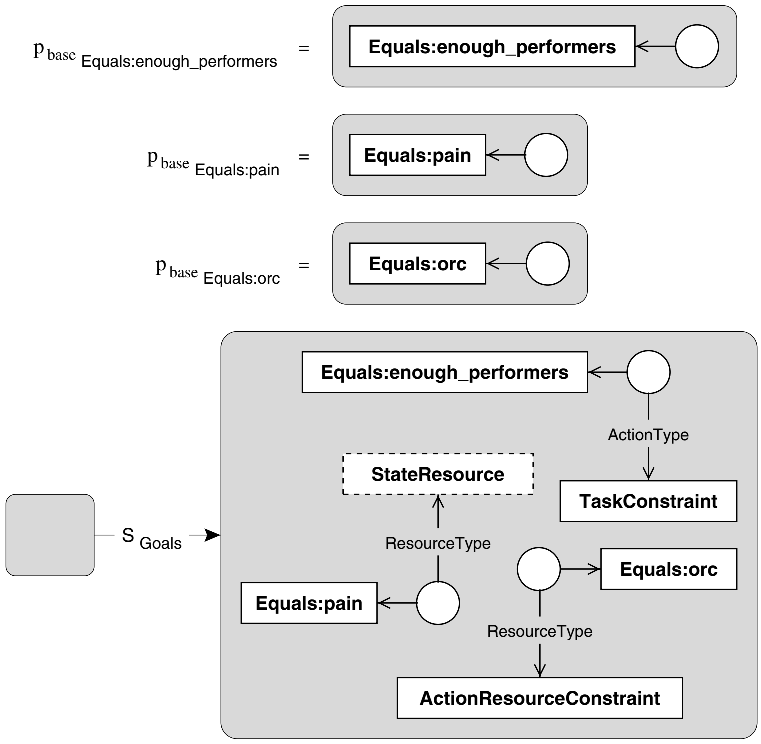


Fig. 5.2. Specification of S_{Goals} for the Orc Quest Example

– delete a `catch_a_group` task constraint
– add a `deliver_humans` task constraint
– delete a `deliver_humans` task constraint

For all six decision alternatives, there is a constraint-internal value that represents the preference for this alternative. All preference values are initially set to 1. If a constraint is called to effect an improvement of the plan, it increases an alternative's value by one if this alternative was chosen by the constraint's last improvement decision and the constraint's goal-/cost-function value is now better than the last time the constraint was called. If the goal-/cost-function value has deteriorated or is the same as last time, the preference value is decreased by two. Each time there is a consecutive deterioration, the decrease is doubled. However, no preference value can fall below one. Then, an alternative is chosen with a probability that is proportional to the alternative's preference value[3].

In the following, the heuristic is illustrated using the Pain State Resource Constraint. Consider a situation in which the current plan consists

[3] See Appendix C for an improved utility measure.

of 5 `catch_only_one` actions (C1A), 10 `catch_a_group` actions (CGA) and 3 `deliver_humans` actions (DHA) and the Pain constraint is chosen to perform a plan improvement (see Situation 1 in Fig. 5.3).

	Current Plan			**Pain State Resource Constraint**							
				Goal Function Value	**Preference Values**						
	#C1A	**#CGA**	**#DHA**		**+C1A**	**-C1A**	**+CGA**	**-CGA**	**+DHA**	**-DHA**	**Penalty**
Situation 1	5	10	3	12	1	2	1	3	3	1	2
					1	2	1	3	4	1	2
					select +DHA						
	5	10	4								
...											
Situation 2	5	14	4	17	1	2	1	3	4	1	2
					1	2	1	3	2	1	4
					select +DHA						
	5	14	5								
...											
Situation 3	8	16	5	17	1	2	1	3	2	1	4
					1	2	1	3	1	1	8
					select -CGA						
	8	15	5								
...											
Situation 4	11	15	5	16	1	2	1	3	1	1	8
					1	2	1	4	1	1	2

Fig. 5.3. The Heuristic of the Pain State Resource Constraint

The Pain constraint's goal function calculates the sum of the *Contribution* variables of the linked state tasks. There are currently 5 linked state tasks with a *Contribution* of 1, 10 with a *Contribution* of 4, and 3 with a *Contribution* of -11. Thus, the constraint's goal function has a value of $5 \times 1 + 10 \times 4 + 3 \times (-11) = 12$. The current preference values of the constraint are given in Fig. 5.3 ("+" means addition and "-" means deletion).

Let's assume that the constraint's goal function value is now better than the last time the constraint was called and that the constraint's last improvement decision was to add a `deliver_humans` action. Accordingly, this decision's preference value is rewarded by increasing it by one (next line in Fig. 5.3). Next, an alternative to improve the current situation is selected. The choice probability for the +DHA option is the highest (preference value divided by the sum of all preference values; $4/12 = 33\%$), and we assume that this alternative is chosen. The plan/CSP is changed according to this option.

After some iterations, the Pain constraint can be called again (see Situation 2 in Fig. 5.3). In the meantime, the plan has been changed and some more `catch_a_group` actions are added. The Pain constraint's goal function value has deteriorated since the last time the constraint was called and its last decision's preference value is therefore decreased by the penalty value of 2. In addition, the penalty value is doubled. The choice probability for the +DHA option is now only $2/10 = 20\%$, but we assume that this alternative is chosen again. The plan is changed according to this option.

After some time, the Pain constraint is called once more (Situation 3). The constraint's goal function value is the same as at the last call. Stagnation is considered to be as bad as deterioration, and the +DHA option's preference value is therefore decreased by the penalty value of 4, and the penalty value is doubled again. However, the +DHA preference value is increased to its minimum of 1 to ensure that the option retains a chance of being chosen. We assume that the -CGA alternative is chosen this time (probability of $3/9 = 33\%$). The plan is changed according to this option.

At the next call of the Pain constraint (Situation 4), the constraint's goal function value has improved, and so, the -CGA preference value is increased and the penalty value is set back to 2.

5.1.3 The Global Search Control

Within the global search control, the Performers constraint will always be chosen to perform an improvement if the plan does not yield at least 5 performers. Otherwise, if the plan yields any pain, the Pain constraint will be chosen. If we have a plan with enough performers and no pain, the Orc constraint will be chosen to help shorten the plan. This enables a search to be realized that has the performers criterion as its satisfaction goal, the pain criterion as the primary optimization goal and the duration criterion as the secondary optimization goal.

The search process is initialized with a CSP that includes the Performers, Pain and Orc constraint and a Task Constraint with an *ActionType* variable that represents the task constraint's fourth configuration (to establish the satisfaction goal `Performers` $\geq$ `5`). The state resource constraints Performers and Pain are initialized with *CurrentState* values of 0.

5.1.4 Evaluation

If the planning system is started, the shortest possible plan that results in no pain for the Orc can quickly be obtained (55 times `catch_only_one` and 5 times `deliver_humans`, which yields a plan duration of **115 hours**, a **0 on the pain scale** and **5 performers**). Figure 5.4 shows the temporal distribution for 100,000 test runs with different random seeds (e.g., after 1,000 improvement iterations, 100 % of the test runs found a plan that yields enough performers, 94 % of the test runs found a plan that yields enough performers and no pain, and 41 % of the test runs found the shortest plan that yields enough performers and no pain).

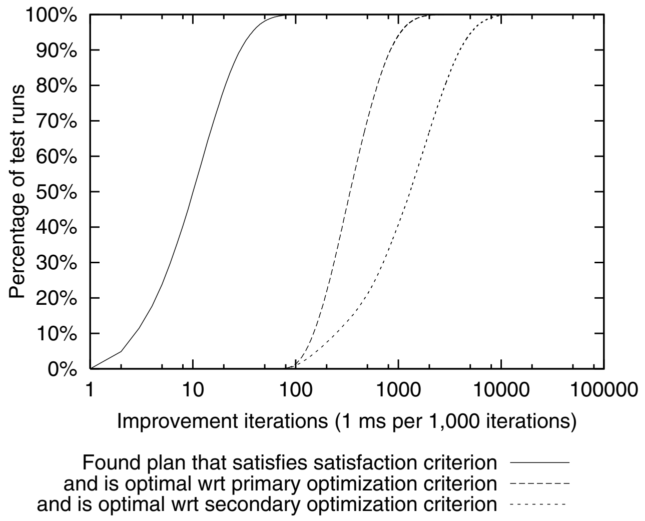


Fig. 5.4. Test-Run Results for the Orc Quest Example

A comparison to other systems is not very useful, as these would either come up with the plan that primarily minimizes the plan length (`catch_a_group & catch_a_group`) or require a bound for the plan length. If we cheat in favor of the other planning systems and provide the information that the optimal length bound is 60 actions, an encoding according to CPlan [175] of this simple problem would require 129,600 variables[4] – not to speak of constraints! Even the time required to create a list of these domain variables (tested with CHIP, which is a state-of-the-art constraint programming system) is longer than any of the test runs in Fig. 5.4 needed to compute the optimal plan. However, we actually cheated in favor of CPlan by providing the optimal length bound. In general, the comparison is not a quantitative question of speed but a qualitative question of *capability* to find the optimum.

[4] **60** steps $\times$ (duration domain **60** $\times$ **6** + pain domain **60** $\times$ **16** + performers domain **60** $\times$ **14**) = **129,600**.

However, if one insist on quantitative results, it is a matter of improving the heuristics. The conceptual approach of the planning system allows us to easily integrate domain-dependent knowledge. For example, the Pain constraint can be improved by paying more attention keeping the same number of performers if an option to reduce the pain is chosen. For this purpose, three additional `catch_only_one` actions can be added if the option to delete a `catch_a_group` action is chosen. In the same way, the number of performers can be replenished by adding `catch_only_one` and `catch_a_group` actions (partitioning them according to the preference values) if the option of adding a `deliver_humans` action is chosen. These simple modifications already make it possible to considerably reduce the execution times (see Fig. 5.5).

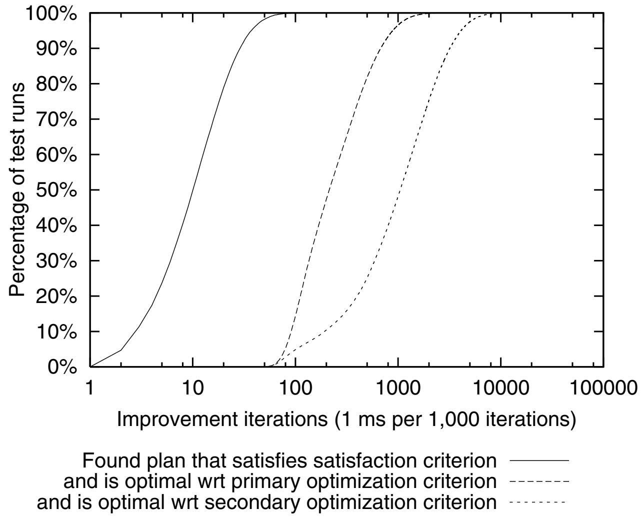


Fig. 5.5. Test-Run Results for the Orc Quest Example (II)

Because of the logarithmic scales of Fig. 5.4 and 5.5, the improvement is not very evident, but the average number of iterations needed until the optimal plan is found is reduced by 20 %.

5.2 Domain-Independent Planning

The constraints used in the previous section work well for the Orc Quest example, but are not generally applicable. A preference value for each possible modification is not feasible for more complicated resources. Also, precondition requirements and temporal aspects were neglected. This section presents more general constraints, which can be used for a wider range of problems. Of course, the conceptual approach still makes it possible to easily integrate domain-specific knowledge for specific domains.

Sections 2.4.1 and 2.4.1 already presented solutions for an Action Resource Constraint and a Task Constraint. They were developed for the job-shop scheduling domain but can be applied here as well. A small extension is made to the constraints, which is described in Sect. 5.2.1.

The most important constraint missing for the planning domain is the State Resource Constraint. In Sect. 5.2.2, a State Resource Constraint is presented that features a symbolic state domain (like the Location SRC of Fig. 4.4). This enables most of the common benchmark domains to be handled, e.g., problems specified in the STRIPS formalism.

The resulting system[5] is evaluated in Sect. 5.2.3.

5.2.1 Extending the ARC and TC

The Action Resource Constraint was introduced in a framework without any possibility of modifying the graph structure. Thus, the potential being extended by the structural constraint satisfaction, the constraint's heuristics can use additional techniques to reduce their inconsistency. A simple extension made here is to include an ARC-H3 heuristic that deletes an action if an Action Task of the action causes an overlap on the resource.

The heuristic selects an Action Task, the choice probability for an Action Task being proportional to the length of all ARC intervals that include the task and are inconsistent. The Task Constraint of the selected Action Task is deleted (together with its tasks – in the same way as in Appendix D.2).

The planning system will be used in the context of an ongoing time. To force the planning system to schedule Action Tasks that have not yet been started to begin after the Current Time, the Task Constraint is extended to assign inconsistencies to these Action Tasks. The inconsistency is computed as the temporal distance from the Action Task's *Begin* variable to the Current Time's variable.

No heuristic allows a deletion or a move of an Action Task that is currently under execution.

5.2.2 A State Resource Constraint with a Symbolic State Domain

This section describes a simple version of a State Resource Constraint with a symbolic state domain. More specifically, the constraint has the following features:

[5] The system has been implemented in the language ObjectC and consists of more than 10,000 lines of code. However, the implementation is very prototypical and many routines are implemented in a manner that they can be easily changed instead of maximizing efficiency.

– The SRC has a finite number of possible states S.
– The *TemporalReference* variables of connected PRECONDITION TASKs have a domain of integers from 0 to a planning horizon h, and the PRECONDITION TASKs' *State* variables have a domain of S.
– The *TemporalReference* variables of connected STATE TASKs have a domain of integers from 1 to h.
– The SRC has a finite number of possible events that can cause a state change.
– The SRC's synergy mapping is a partial function which maps a set of STATE TASKs' *Contributions* to an event.
– The SRC's state transition is a partial function which maps a state and an event to a successor state.
– The state is projected on a timeline of discrete time points from 0 to h according to the connected STATE TASKs and the initial value of the CURRENT STATE. If a SENSOR CONSTRAINT is connected to the SRC's OBJECT, the state at the CURRENT TIME c is set to the CURRENT STATE, regardless of whether there is a valid state transition or not.
– For the state projection, the present state is preserved if it is not possible to derive an event from the set of all STATE TASK *Contributions* at that time point by the synergy mapping, or if no state transition is possible from the SRC's present state for the derived event.
– If the state required by a connected PRECONDITION TASK is not equal to the SRC's actual state at this time point, the SRC's costs are increased by the minimal number of state transitions required to transform the actual state to the required state.

The constraint's internal structures and heuristics are detailed in the following sections.

Internal Structures. The structures used in the current implementation are extremely complex in order to provide fast access to heuristic guidance information and to minimize effort in the case of changes. For this reason, only the main structures are described here.

As for the ACTION RESOURCE CONSTRAINT, a list of multiple-linked temporal intervals forms the constraint's basic structure (see Fig. 5.6). The timespan from 0 to h (the planning horizon) is covered by the intervals. For every distinctive *TemporalReference* of a linked STATE TASK, a new interval is started. If a SENSOR CONSTRAINT is connected to the SRC's OBJECT, a new interval starts at c as well (and the value of the CURRENT STATE's variable becomes the interval's state).

Every interval stores links to the STATE TASKs that are located at the interval's begin (and are thus responsible for the interval's existence). In addition, an interval stores a link to every PRECONDITION TASK that is within the range of the interval, together with a value that expresses the inconsistency of the PRECONDITION TASK. The inconsistency of a PRECONDITION TASK is given by a state-distance table, which provides a distance value for

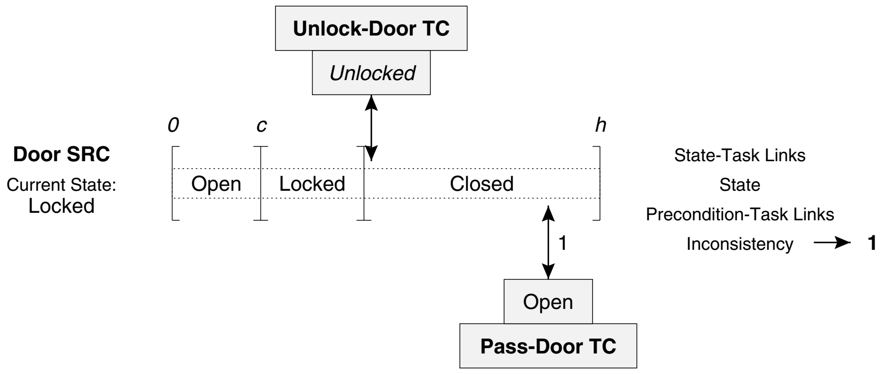


Fig. 5.6. An SRC's Interval Structures

every possible transition from one state to another. The distance value that is considered for the inconsistency of a PRECONDITION TASK is the value for the transition from the interval's state to the state requested by PRECONDITION TASKS. The SRC's costs are computed by the sum of the PRECONDITION TASKS' inconsistencies.

The state-distance table is computed by calculating the minimal number of state transitions needed to get from one state to another (see Fig. 5.7). If it is not possible to get from one state to another, a special *dead-end* value is entered, which is higher than the maximal distance value.

Possible Transitions

Open	&	*Closed*	→	Closed
Closed	&	*Opened*	→	Open
Closed	&	*Locked*	→	Locked
Closed	&	*Smashed*	→	Smashed
Locked	&	*Unlocked*	→	Closed
Locked	&	*Smashed*	→	Smashed

State-Distance Table

↗	Open	Closed	Locked	Smashed
Open	0	1	2	2
Closed	1	0	1	1
Locked	2	1	0	1
Smashed	100	100	100	0

Fig. 5.7. An SRC's State-Distance Table

Selecting an Improvement Heuristic. To select one of the constraint's improvement heuristics, the general applicability of the heuristics is first checked (see following sections for applicability conditions). Only applicable heuristics are considered for the following choice process.

The choice of a heuristic is made in exactly the same way as the choice of a modification alternative in the Orc Quest example (with the improved utility measure described in Appendix C). The chosen heuristic is applied.

An applied heuristic can return a negative *success value*, which means that the heuristic was unsuccessful and no change has been made. In this case, the choice process is restarted. This restart means that the usual update of the preference values is skipped. Instead, the failed heuristic's preference value is temporarily divided by two (though no value may fall below one) because the probability that the heuristic will return a negative success value again is higher than before. If one of the improvement heuristics has been successfully applied, all adaptations of the preference values that were done for the restarts are undone.

SRC-H1: Adding an Event. This heuristic is always applicable. The idea is to introduce an event that changes the SRC's state in such a way that a Precondition Task becomes less inconsistent.

Selecting an Inconsistent Precondition Task

One of the SRC's inconsistent Precondition Tasks that is after the current time c is selected with a choice probability for a Precondition Task that is proportional to the inconsistency caused by the Precondition Task.

Selecting an Interval for the Event Insertion

An interval i is to be chosen, in which an event will be added to reduce the inconsistency of the chosen Precondition Task p. Initially, i is set to the interval of p. If p is at the very start of i, the heuristic is stopped and a negative success value is returned, because this means that there is no room to include an event without its overlapping with the interval's event.

Each time the state-distance table provides the information that it is not possible to transform the state of i into the state requested by p, i is set to its predecessor interval. If i is the first interval and a state transition is still not possible, the heuristic is stopped and a negative success value is returned. This is also done if a state transition from i's state to p's requested state is possible but i is not long enough for an event to be inserted without causing an overlap with the event of i.

Selecting a Time Point for the Event Insertion

The event is to be placed at a time point t, which is to be between two previously computed time points t_1 and t_2. The idea of this balanced placement is to ensure a maximal space with respect to other tasks that interact with the event and with further events that would need to be inserted to achieve full satisfaction of p.

If i is still the interval of p, t_1 is computed by setting it to the same time point as the interval's last Precondition Task before or at the same time point of p. If i has no Precondition Tasks or all other Precondition Tasks of i come after p, t_1 is set to the interval's begin. If t_1 is before c, t_1 is set to c.

If i is before the interval of p, t_1 is computed by setting it to the same time point as i's last Precondition Task. If i has no Precondition Tasks, t_1 is set to the i's begin.

t_2 is set to the same time point as p. If i is before the interval of p, t_2 is set to the last time point of i. To place t at a time point such that there is enough room for further events that would have to be inserted to achieve full satisfaction of p, t is computed as:

$$t \quad = \quad round\left(t_1 + \frac{(t_2 - t_1)}{1 + statedistance(state(i) \rightarrow state(p))}\right)$$

There is a possibility of t_1's being set to an interval's begin. This placement would be unfavorable because a new State Task at this time point may interact with the interval's State Tasks. Thus, if t is equal to t_1, t is set to $t+1$. If t is now higher than t_2, t is set back to t_2. If t is before c, the heuristic is stopped and a negative success value is returned.

Selecting an Event to Be Added

One state is chosen from among all the states that can be derived by a state transition from i's state, a choice probability for a state being proportional to $\frac{1}{d^3}$, where d is the state's state-distance to the state required by p. The event that causes the selected state is the event to be added.

Adding the Event

If multiple different sets of *Contributions* can cause the event, one of the sets is chosen at random. The addition of a new action (i.e., of a Task Constraint) described below is done for every *Contribution* of the selected set.

There may be numerous different *ActionTypes* for Task Constraints that imply a State Task with the required *Contribution.* To choose from among them, the insertion of each relevant type of Task Constraint (together with its tasks – in the same way as in Appendix D.2) is simulated, the change in overall inconsistency being recorded. The tasks of an action are inserted at time points such that the required *Contribution* occurs at time point t, and the temporal relations of the Task Constraint are ensured by a process similar to that described in Sect. 2.6.2.

The choice between the *ActionTypes* is made according to Appendix C.5 in such a way that the choice probability is dependent on the inconsistency change. The chosen action is inserted as in the simulation.

SRC-H2: Moving an Event. This heuristic is applicable if at least one State Task is connected to the SRC. The idea is to temporally move an event to a time point before a Precondition Task such that the Precondition Task becomes less inconsistent.

Selecting an Inconsistent PRECONDITION TASK

One of the SRC's inconsistent PRECONDITION TASKs that is after the current time c is selected with a choice probability for a PRECONDITION TASK proportional to the inconsistency caused by the PRECONDITION TASK.

Selecting an Event to Be Moved

For the inconsistency improvement of the chosen PRECONDITION TASK p, the events of the predecessor interval i_p and the successor interval i_s of p's interval i are considered. The predecessor option is dropped if i or i_p are the first interval or if they begin before c. The successor option is dropped if i is the last interval.

For the predecessor option, a state s_p is computed, which is the state that results from a state transition from the state of the predecessor interval of i_p using the event of i, and a subsequent state transition from the resulting state using the event of i_p. If one of the state transitions is not possible, the predecessor option is dropped.

For the successor option, a state s_s is set to the state of i_s.

For each option, an improvement value Δ_p/Δ_s is computed as the state distance from the option's state s_p/s_s to the state of p, minus the state distance from i's state to that of p. If one of the computed improvement value is less or equal to zero, the corresponding option is dropped.

From all the options, an interval is chosen with a choice probability for an interval that is proportional to $\frac{1}{d^3}$, where d is the improvement value Δ_p/Δ_s. If there are no options for the choice, the heuristic is stopped and a negative success value is returned.

Moving the Event

The selection of a time point for the event placement is done in exactly the same way as SRC-H1's selection of a time point for the event insertion. All of the chosen interval's STATE TASKs are moved to the selected time point.

SRC-H3: Moving a Precondition. This heuristic is applicable if there is at least one inconsistent PRECONDITION TASK that is not connected to the TASK CONSTRAINT of S_{Goals}. The idea is to temporally shift a PRECONDITION TASK such that its inconsistency is decreased.

Selecting an Inconsistent PRECONDITION TASK

One of the SRC's inconsistent PRECONDITION TASKs that are not connected to the TASK CONSTRAINT of S_{Goals} is selected with a choice probability for a PRECONDITION TASK that is proportional to the inconsistency caused by the PRECONDITION TASK. If one of the ACTION TASKs that belong to the chosen PRECONDITION TASK's TASK CONSTRAINT has already been executed or is currently under execution, the heuristic is stopped and a negative success value is returned.

Selecting an Interval

The new places considered for the chosen PRECONDITION TASK p are in the predecessor interval i_p or successor interval i_s of p's interval i. The predecessor option is dropped if i is the first interval. The successor option is dropped if i is the last interval.

Each time the state-distance table provides the information that it is not possible to transform the state of i_p into the state requested by p, i_p is set to its predecessor interval. If i_p is the first interval and a state transition is still not possible, then the predecessor option is dropped.

Each time the state-distance table provides the information that it is not possible to transform the state of i_s into the state requested by p, i_s is set to its successor interval. If i_s is the last interval and a state transition is still not possible, then the successor option is dropped.

If p's inconsistency is not equal to the dead-end value, p's current interval i is also an option, i.e., no change will be made if i is chosen.

From all interval options, an interval is chosen with a choice probability for an interval that is proportional to $\frac{1}{d^3}$, where d is the state distance from the interval's state to the state required by p. If there are no interval options for the choice, or if i is the chosen interval, the heuristic is stopped and a negative success value is returned.

Moving the PRECONDITION TASK

If the chosen interval is the last interval, the PRECONDITION TASK p is moved to the interval's beginning. Otherwise, the PRECONDITION TASK p is moved to the middle of the chosen interval.

SRC-H4: Deleting an Event. This heuristic is applicable if at least one STATE TASK is connected to the SRC. The idea is to improve the inconsistency of a PRECONDITION TASK by deleting a STATE TASK preceding it.

Selecting an Inconsistent PRECONDITION TASK

One of the SRC's inconsistent PRECONDITION TASKs is selected with a choice probability for a PRECONDITION TASK that is proportional to the inconsistency caused by the PRECONDITION TASK. If the chosen PRECONDITION TASK belongs to a TASK CONSTRAINT one of whose ACTION TASKs has already been executed or is currently under execution, the heuristic is stopped and a negative success value is returned. This is also done if the interval of the PRECONDITION TASK is the first interval.

Deleting a STATE TASK

One of the interval's STATE TASKs is chosen at random. The TASK CONSTRAINT of the selected STATE TASK is deleted (together with its tasks – in the same way as in Appendix D.2).

SRC-H5: Deleting a Precondition. This heuristic is applicable if there is at least one inconsistent PRECONDITION TASK that is not connected to the TASK CONSTRAINT of S_{Goals}. The idea is to delete an action if it is very hard or impossible to fulfill its precondition.

Selecting an Inconsistent PRECONDITION TASK

One of the SRC's inconsistent PRECONDITION TASKs that are not connected to the TASK CONSTRAINT of S_{Goals} is selected with a choice probability for a PRECONDITION TASK that is proportional to the inconsistency caused by the PRECONDITION TASK. If one of the ACTION TASKs that belong to the chosen PRECONDITION TASK's TASK CONSTRAINT is currently under execution or has already been executed, and one of the TASK CONSTRAINT's STATE TASKs still comes after c, the heuristic is stopped and a negative success value is returned.

Deleting the Corresponding TASK CONSTRAINT

The TASK CONSTRAINT of the selected PRECONDITION TASK is deleted (together with its tasks – in the same way as in Appendix D.2).

5.2.3 Evaluation

This section sets out to demonstrate that the resulting planning system is able to handle general domain-independent tasks and produces good results even with the simple SRC described above.

The Logistics Domain. There are lots of example domains in the planning area. For our evaluation, variations of the "Logistics Domain" are chosen because these are highly relevant to computer games. In many strategy and simulation games, resources such as gold, wood, stones or troops must be transported by different types of vehicles.

The Logistics Domain specifies the problem of transporting packages between locations (see Fig. 5.8). Transportation within a city can be done by a truck. But a truck cannot leave its city. Transportation between locations in different cities must thus be done by airplane. An airplane can only serve one specific location of a city (the airport). A truck and an airplane have actions to load a package, drive/fly to a specific location and unload a package. A problem specification includes a set of locations with information about a location's city and whether the location is an airport, the initial positions of all existing trucks, airports and packages, and the target positions for (not necessarily all) packages.

Realization. The problem does not correspond to a typical instance of an agent's configuration (hands, feet, ...), but it is nevertheless easy to model. Every package, truck and airplane has an action resource, which prevents multiple operations from being executed by an object at the same time (like

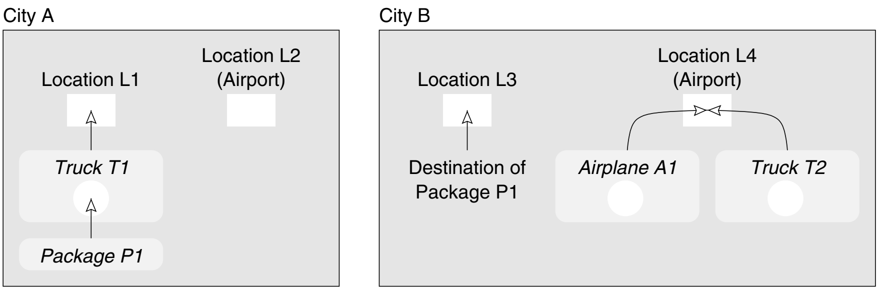


Fig. 5.8. A Simple Example of the Logistics Domain

a truck being driven and simultaneously being loaded with a package). Every package, truck and airplane also has a state resource, which defines its location. The initial locations are stored in the corresponding CURRENT STATES' variables, the CURRENT TIME's variable is set to 0 and the packages' destinations are specified by the S_{Goals}'s TASK CONSTRAINT with corresponding PRECONDITION TASKS. The TASK CONSTRAINT is given all possible action configurations.

The ARC's choice between its improvement heuristics is made with a probabilistic distribution of 90 % for ARC-H2, 9 % for ARC-H1 and 1 % for ARC-H3. This distribution is based on the experiments described in Sect. 2.6.1 and was empirically proved, in this context too, to be superior to other ratios. The SRC's choice of heuristic is self-adapting and does not need to be given any parameter.

The TASK CONSTRAINT's choice of a heuristic is modified so as to always apply only the more aggressive TC-H2 heuristic. This would appear to be more suitable for the planning context because the temporal order here is much more important than for scheduling.

Results. The benchmark problems of the AIPS-2000 planning competition (track 2; see Appendix A.4) are used in the following. Because of the stochastic nature of the planning system, 1,000 test runs were executed per problem setting. This means that it was only possible to analyze small problems (problems 4 to 10 with variations 0 and 1) because of limited computing power. Instead of computation time, the number of iterations was measured, because the test runs were executed on different types of computers (the average being about 400 iterations per second). A test run was stopped if it failed to find a solution after 100,000 iterations.

Figures 5.9 and 5.10 show the number of iterations necessary to find a solution (to facilitate comparison, problem 9-1 is included in both figures). The heavy-tail phenomenon (which roughly means that a small fraction of test runs take very long – see [76] for details) is clearly evident, and a restart technique could be applied to greatly improve average-case behavior.

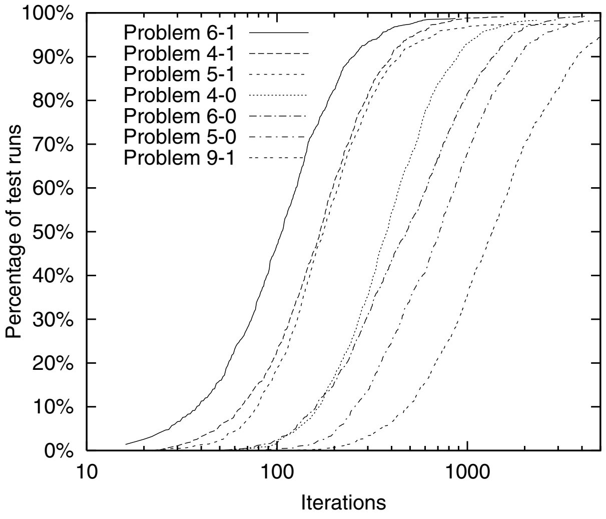


Fig. 5.9. Runtime Results for the Logistics Domain (Easy Problems)

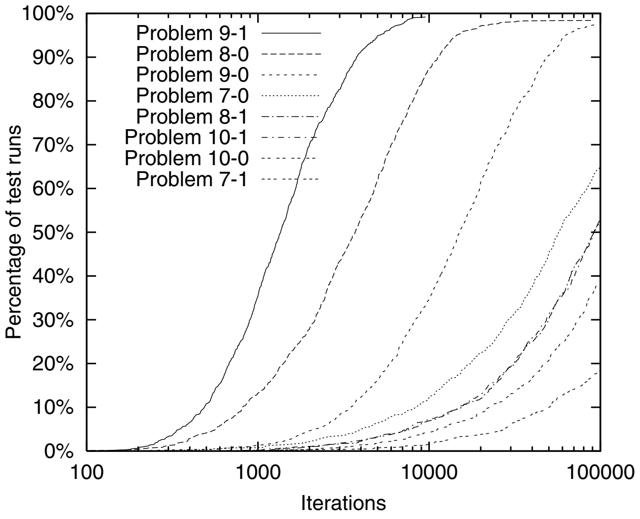


Fig. 5.10. Runtime Results for the Logistics Domain (Harder Problems)

The time taken to solve a problem is closely related to the number of actions necessary to solve the problem (see Fig. 5.11 and 5.12). The "steps" in the graphs can be explained by the fact that it does not normally harm a plan to add an action and then add another action to undo the preceding one, e.g., to load a package and then unload it, whereas the addition of a single action can result in a different outcome. Thus, more test runs will yield results with two additional steps instead of one.

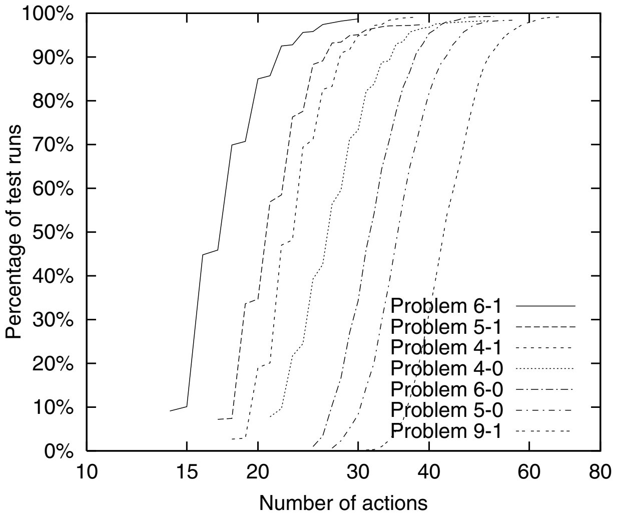


Fig. 5.11. Plan-Step Results for the Logistics Domain (Easy Problems)

Tabu Lists. Even without applying domain knowledge, there are an endless number of ways of improving the results. At least one simple way is tried below – tabu lists.

One possible way of applying a tabu list is for the global search control's constraint selection. This failed, however, in the job-shop-scheduling experiments (see Sect. 2.5.3) and will not be tried here again.

Another possibility is to use a tabu list that stores the objects affected by changes (e.g., an Action Task moved by ARC-H2 or a Task Constraint inserted by SRC-H1). If no new object is stored in the list within an iteration (or in the case of unsuccessful application of an SRC's heuristic), an empty element is stored in the list to guarantee an ongoing replacement of the elements. A heuristic that tries to modify one of the list's objects fails to do so. Figure 5.13 shows the results for Problem 6-1.

Applying long tabu lists results in a great initial improvement. With an increasing runtime, shorter/no tabu lists prove to be better (which is irrelevant for an approach of rapid restarts). However, the use of tabu lists yields plans

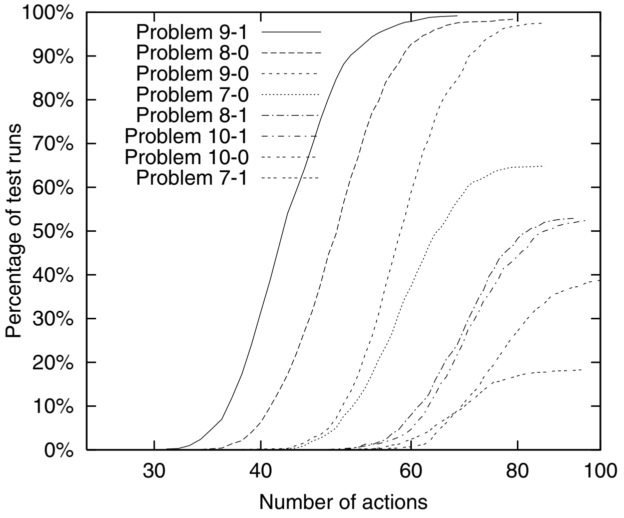


Fig. 5.12. Plan-Step Results for the Logistics Domain (Harder Problems)

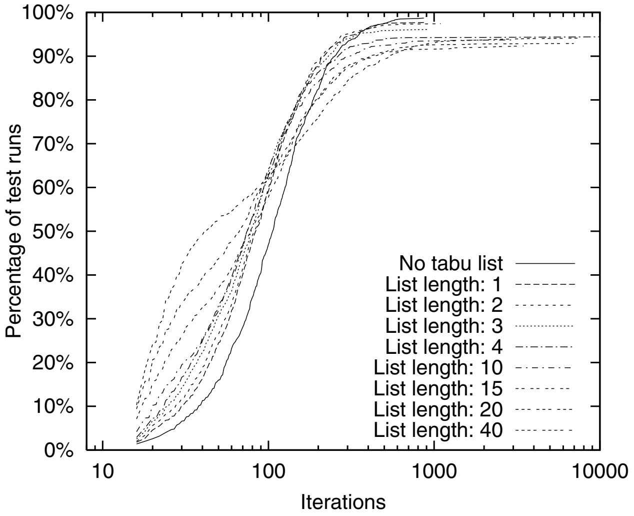


Fig. 5.13. Tabu-List Application on Problem 6-1

with a large number of actions (Figure 5.14 shows the results for Problem 6-1).

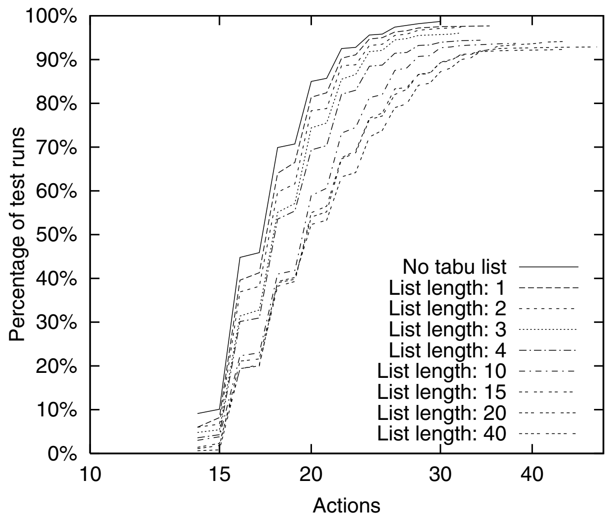


Fig. 5.14. Plan-Step Results for the Tabu-List Application

Many other "standard" methods for achieving an improvement could be tried, but this is not our main concern here.

Enhancing the Problem Domain. For a computer-games application, numerous other issues must be considered. For example, it takes a certain time to get to another location, and the time it takes to deliver the resources/goods should be minimized. These problems are addresses in the following sections.

Durations. The planning system inherently provides a temporal projection. Thus, enhancing the actions by adding a duration is no problem at all. Indeed, the previous problems were already treated as temporal planning problems – each action having a length of one time unit. Of course, this can be changed, Figure 5.15 showing the results for three different modifications[6] of Problem 6-1, in which every type of ACTION TASK has a duration of 100 plus a random value of between -99 and +100. It is obvious that taking durations into account has no effect on runtime.

Optimization. So far, only satisfaction goals have been considered. However, the construction of a plan – using *domain knowledge* – could actually be accomplished in *linear time* by adding the necessary actions for one package after the other. Even using a very unsophisticated approach, 12 actions at

[6] The detailed specification of the modified problems is available at the EXCALIBUR project's webpage.

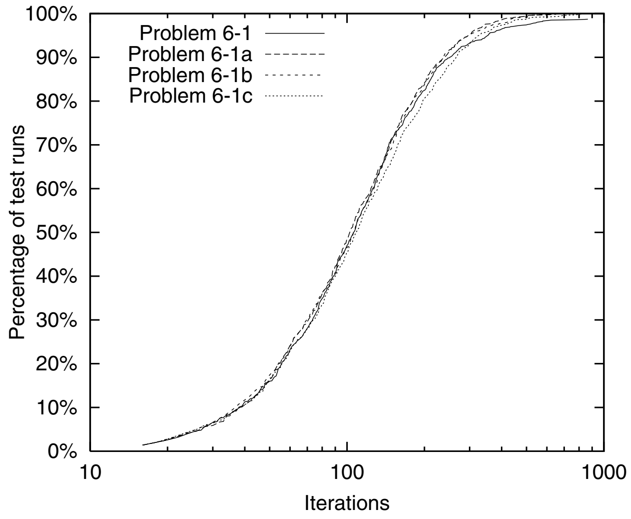


Fig. 5.15. Duration Enhancements for Problem 6-1

the most would be necessary per package: a truck is driven to the package's location, the package is loaded on the truck, the truck is driven to the airport, the package is unloaded, an airplane is flown to the airport, the package is loaded on the airplane, the airplane is flown to the destination city, the package is unloaded, a truck is driven to the airport, the package is loaded on the truck, the truck is driven to the package's destination, and the package is unloaded.

If a minimization of the total delivery time (i.e., minimizing the maximal completion time of the single deliveries) is desired, a goal function can be incorporated into the SRC that returns the duration from 0 to the *Begin* of the PRECONDITION TASKs of the S_{Goals}'s TASK CONSTRAINT. An improvement heuristic for this goal function can be implemented very simply: it shifts the PRECONDITION TASKs in the direction of the current time until the first time point is reached at which the SRC's inconsistency increases.

The question now is the balance between satisfaction and optimization (see also Sect. 3.7.4). Here, the same hierarchical approach as in the Orc Quest example is taken, i.e., optimization of the goal function is only started if the cost function reaches 0.

The initial time point for the *Begin* variables of the PRECONDITION TASKs of the S_{Goals}'s TASK CONSTRAINT is not really important as long as it allows enough time for a consistent solution. For the following experiments, it is set to $NumberOfPackages \times MaximallyNecessaryActionsPerPackage \times MaximalActionLength$, i.e., for the 6-1 modifications, $6 \times 12 \times 200 = 14400$. Figure 5.16 presents three sample test runs for Problem 6-1a, in which the development of the duration of consistent solutions can be seen. The runtime was set to 10,000 iterations for these test runs. The dips indicate an application of the SRC's goal heuristic, while the plateaus indicate a search for a new consistent solution.

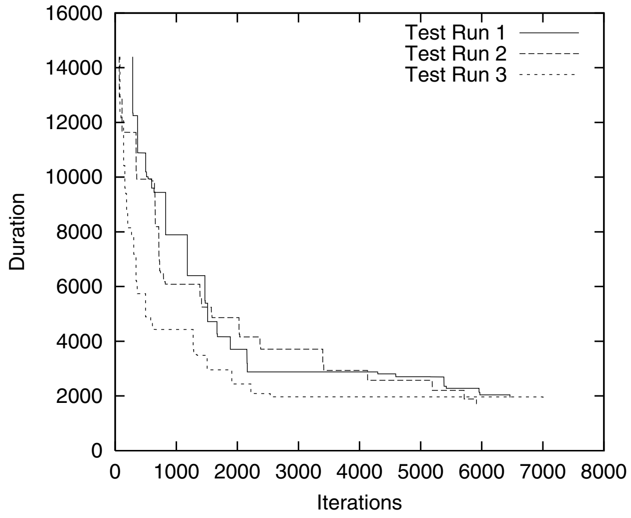


Fig. 5.16. Three Sample Test Runs for Problem 6-1a

For Problems 6-1a, 6-1b and 6-1c, Figure 5.17 shows how many test runs found a certain duration for a runtime of 100,000 iterations. The shortest durations found are 1,150 for 6-1a, 1,004 for 6-1b and 1,177 6-1c.

Again, the use of tabu lists can yield a significant improvement (Figure 5.18 shows the results for Problem 6-1a). However, in contrast to the previous application of tabu lists, the number of possible actions is restricted here because of the optimization performed. This causes that tabu lists of a longer length do not improve the results anymore – there is even a deterioration in the results in contrast to shorter lists.

Dynamics. The feature of being able to handle dynamic changes of the problem specification is also very important for the computer-games domain because goods, their destinations and transportation facilities vary over time.

To test the approach's ability to handle dynamics, a random package is canceled every 1,000 iterations (no matter if it is already on the way or has been delivered) and a new package is inserted (its initial location and destination being decided at random). The time point for the *Begin* variables of the PRECONDITION TASKs of the S_{Goals}'s TASK CONSTRAINT is increased by $MaximallyNecessaryActionsPerPackage \times MaximalActionLength + 1$, i.e., for the 6-1 modifications, by $12 \times 200 + 1 = 2401$, to ensure the existence of a satisfiable plan. Figure 5.19 shows the results for Problem 6-1a. The

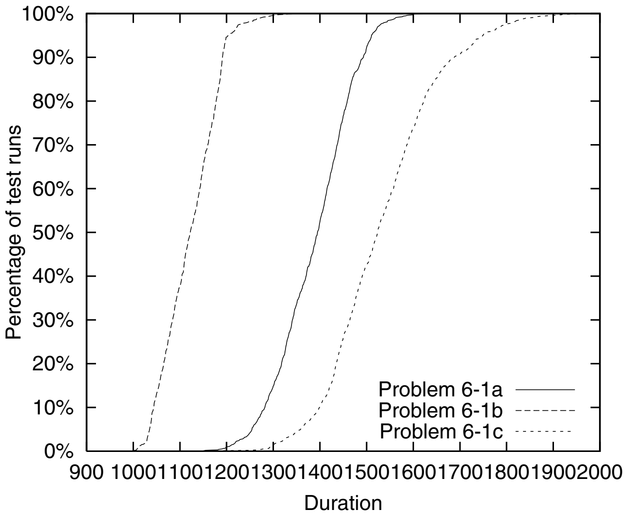


Fig. 5.17. Durations Found for Problems 6-1a, 6-1b and 6-1c

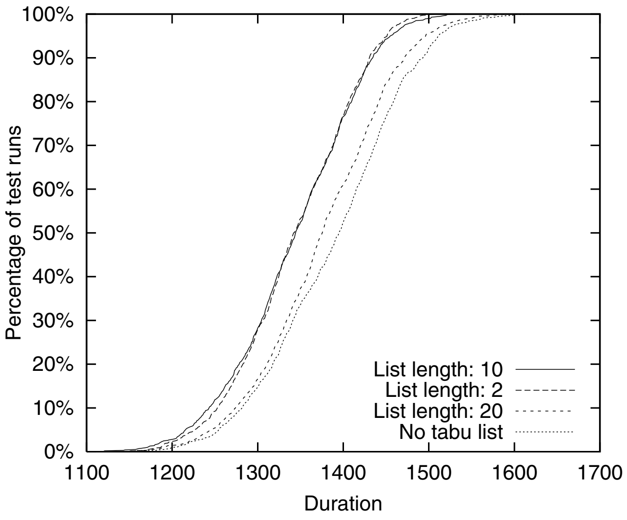


Fig. 5.18. Durations Found for Problem 6-1a Using Tabu Lists

optimization can quickly recover after a change takes place (Figure 5.20 shows the view from above).

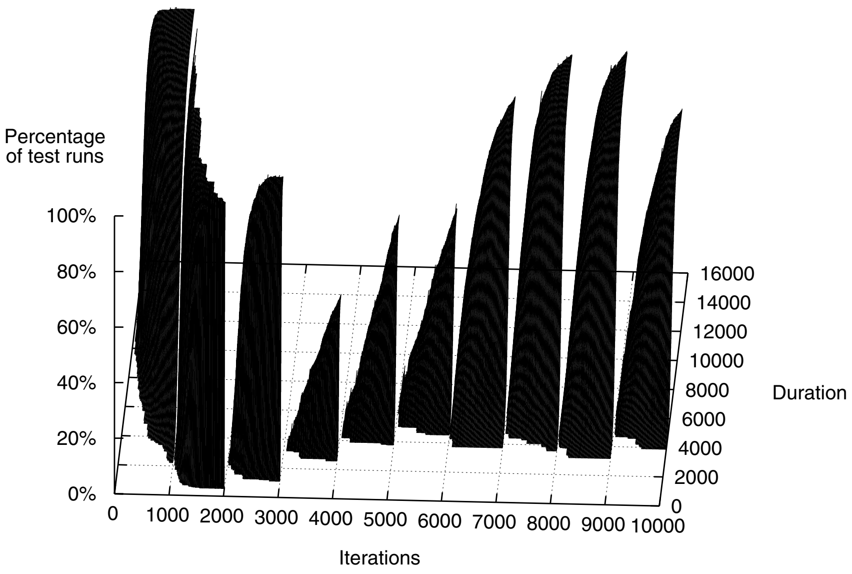


Fig. 5.19. Introducing Dynamics for Problem 6-1a

Figures 5.21 and 5.22 contrast cuttings of the dynamic solution with test runs in which the computation was restarted from scratch for a change. It is clear that the dynamic adaptation converges much more quickly. Larger problems and higher dynamics can mean that a problem cannot even be tackled by a recomputation from scratch because there is not enough time for a regeneration of the problem's representation[7].

Other Extensions. Many other extensions are useful and necessary for the computer-games domain. They include the possibility of introducing new resources (e.g., by building new units to carry goods) and taking into account uncertainty with respect to the delivery's execution (e.g., certain routes are dangerous). Example applications of this are not included. It should, however, have become clear in previous chapters that solutions to these problems can potentially be realized by the techniques presented.

[7] The time for generating the problem representation is not measured in the test runs here; only the number of improvement iterations is considered.

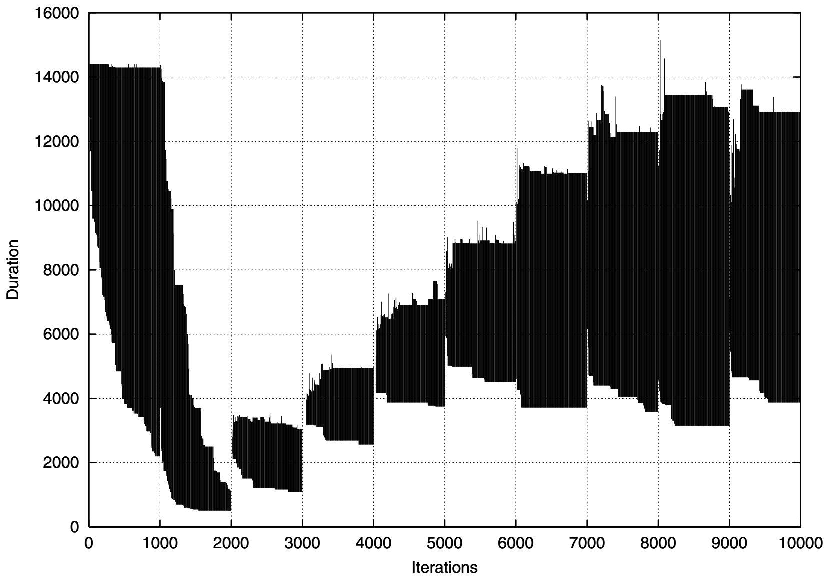


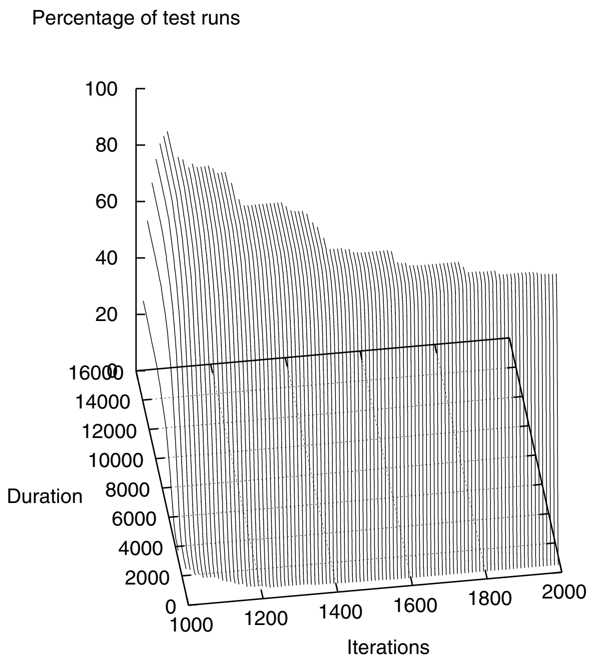
Fig. 5.20. Top View

5.3 Conclusion

This chapter has shown that a system based on the techniques described in the previous chapters can also be applied to handle domain-independent planning tasks, even though the underlying techniques are specifically designed to promote a search guided by domain-dependent knowledge. The very fast specialized solution for the Orc Quest example, as well as the promising results for domain-independent planning tasks, indicate the strength of the underlying technology.

A simple version of a State Resource Constraint featuring a symbolic state domain was presented. Much more complex SRCs will be necessary to tackle more sophisticated problems, e.g., SRCs with state domains of integer numbers, real numbers or even sets (as shown in Fig. 4.42), SRCs with enhanced temporal projections like continuous change (see Fig. 4.3), and SRCs with enhanced support of precondition checks like state-related or temporal ranges. These extensions are beyond the scope of this book and will be the subject of future research.

The system's approach of looking at the problem from a simplified perspective (of one constraint) to choose a heuristic to improve the overall problem is similar to the way in which some other powerful planning approaches proceed, e.g., the approach of Ephrati, Pollack and Milshtein [52], HSP [20] and Fast-Forward [87]. It has actually proved to be a good approach for solving numerous other problems (see also [106]).



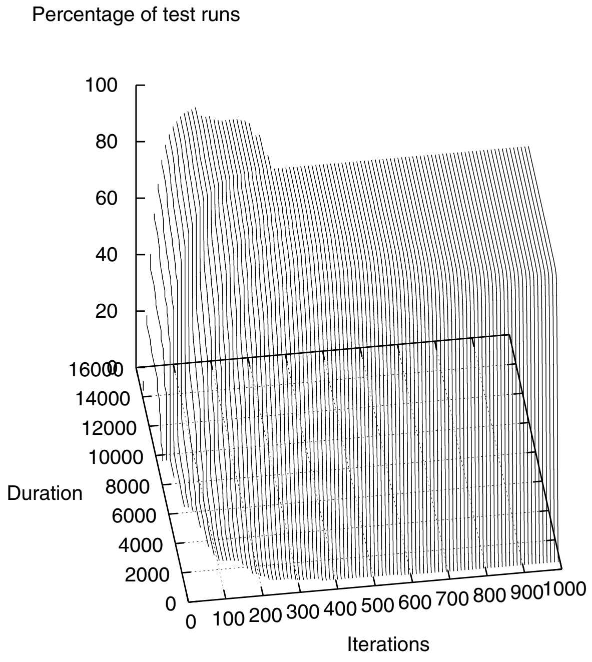
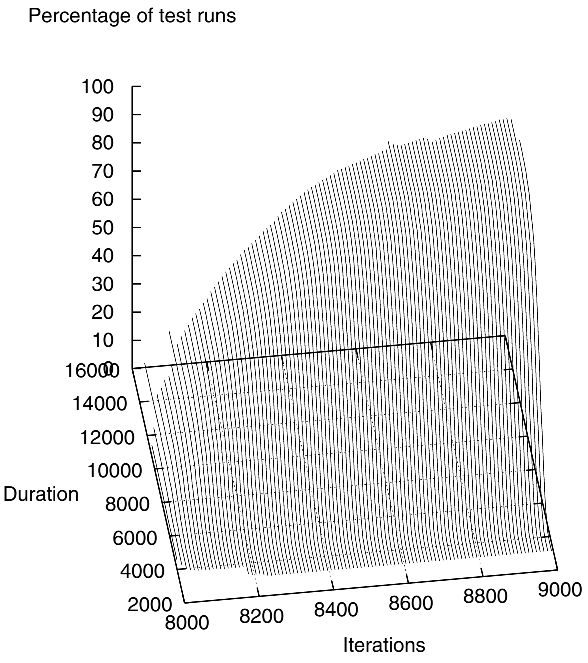


Fig. 5.21. Dynamic Adaptation (Upper) Vs. Recomputation (Lower) – After First Change



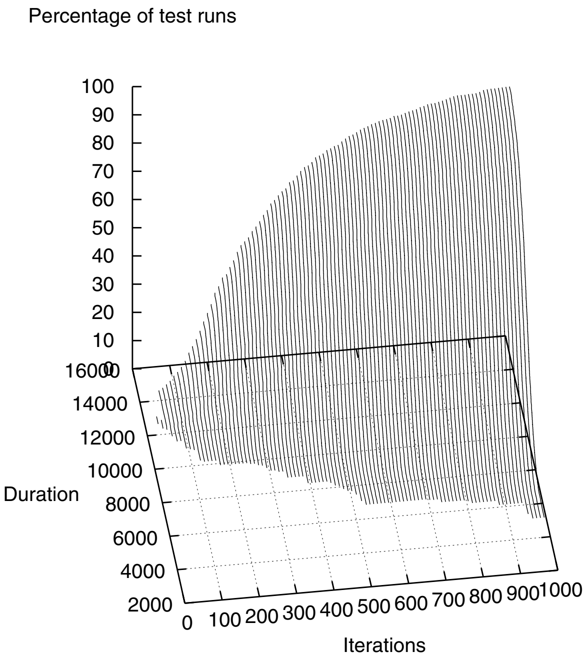


Fig. 5.22. Dynamic Adaptation (Upper) Vs. Recomputation (Lower) – After Eighth Change

The improvement heuristics developed for the constraints have a lot in common with those of the DCAPS/ASPEN/CASPER planning systems [153, 154, 30] and scheduling systems like GERRY [196] and OPIS [167]. The OPIS scheduler is probably the most advanced system with respect to its repair heuristics, as a more sophisticated conflict analysis of the schedules is applied. An integration and extension of these methods for planning tasks seems to be a promising direction for future improvements.

Our system's iterative improvement by local search makes it possible to easily interleave the planning process with sensing and execution. Only a few agent/planning system are able to do this (see also Sect. 1.2). The approach most similar to ours is the CASPER system, which also uses local search strategies to repair a plan. Slightly similar is also the repair strategy of O-Plan [45], tackling broken plans by a set of domain-dependent repair heuristics. A very different approach is applied in CPEF [126], which is based on hierarchical refinement planning and applies a dependency analysis to restart the planning process from the lowest possible hierarchy level.

The chapter included only a limited number of application examples; there are a lot of further empirical studies to be done. Especially regarding problems with an unlimited number of potential resources/objects and with alternatives for sensor anchoring, it will be very interesting to see how much domain-dependent knowledge is necessary to produce reasonable results.

6. Conclusion

The agent architecture presented here focuses on the goal-directed behavior computation of the agent. By contrast, most of the existing architectures for autonomous agents focus on communication protocols and interaction, but do not specify how to compute the behavior of the individual agents (e.g., Reticular Systems' AgentBuilder [156] and SRI International's Open Agent Architecture [114]).

The agent's reasoning about its behavior is based on the paradigm of local search. The iterative repair procedure of local search techniques provides the agent with an efficient anytime reasoning that enables an agent to produce fast reactions as well as sophisticated action plans. Generic agent systems like the Australian Artificial Intelligence Institute's dMARS [46, 66], Bits & Pixels' IaFactory, MASA's DirectIA SDK or the Soar/Games AI engine [176] provide only very simple predefined reasoning schemes. Today's planning systems, on the other hand, focus on deliberative refinement reasoning and are hardly ever able to perform reactive actions. Unlike hybrid systems that involve a reactive and a deliberative module, our agents feature a continuous transition between reaction and deliberation. The priority of a plan's feature to be repaired is given by a cost/goal function. Similar approaches are realized in the GERRY system [196] and ASPEN [31, 154].

Local search's independence of the search history and the quick single repair steps also facilitate an uncomplicated handling of the environment's dynamics with interleaved sensing, planning and execution. This capability is very important for an agent, given the quickly changing environment in computer games.

The whole architecture was embedded in a constraint programming framework, which makes the approach highly declarative and modular and the developed methods applicable to other search problems. However, several extensions of the CP framework were necessary to achieve sufficient efficiency and expressiveness. The first extension realizes an adequate integration of local search. The concept is based on the use of global constraints and preserves the constraint programming framework's properties of declarativeness and variable applicability. Domain-dependent knowledge to guide and accelerate search can be integrated using the global constraints in a plug-and-play manner.

The second extension of constraint programming was necessary to realize a search free from predefined bounds, e.g., plans of a certain length. This feature is important to favor optimization criteria like resource-related properties, and even more important for handling open worlds, in which the agent must reason about an arbitrary number of objects in the world. This was enabled by the concept of structural constraint satisfaction, which enhances conventional constraint satisfaction by providing the opportunity to formulate problems in which the constraint graph is not given in advance. Unlike other planning systems that use productions/rules to change the graph as part of the search (e.g., [7]), modeling planning as an SCSP allows us to specify the problem in a declarative manner and enables the corresponding productions to be deduced automatically. The automatic method guarantees that local-search methods can potentially find all valid plans, which is not normally the case with hand-tailored solutions.

Existing planning systems do not use resource-based criteria as primary optimization goals; instead, most still focus on minimization of the plan length. This approach is quite curious because a plan-length property is irrelevant for real-world problems. Nevertheless, the current planning approaches within search frameworks like OR and SAT also follow this plan-length focus by restricting themselves to the search for plans of a predefined length, expanding this limit if they cannot find a valid plan. This is also true for constraint-based planning approaches like CPlan [175]. Some systems apply separate planning and scheduling phases (e.g., the RETSINA agent [144]), but this prevents them from considering interactions between the decisions regarding planning and resource assignment.

The resource focus is also reflected in the planning model and allows us to use and optimize temporal, spatial and all other kinds of resources. Furthermore, the model is expressive enough to handle incomplete knowledge and information gathering.

To conclude, the presented agent architecture features:

– **Real-Time Reasoning:** The local-search approach enables the system to provide very primitive plans (reactions) for short-term computation horizons, while longer computation times are used to improve and optimize the agent's behavior.
– **Dynamics:** The iterative repair/improvement steps make it possible to easily interleave sensing, planning and execution.
– **Resource Focus:** The search can be conducted in a way that focuses on the satisfaction and optimization of resource-related properties.
– **Incomplete Knowledge:** The model is not restricted to the closed-world assumption and enables an agent's incomplete knowledge to be dealt with.
– **Domain Knowledge:** Even though the model supports full domain-independent planning, domain-specific heuristic knowledge can be easily integrated by way of the global constraints.

– **Software Engineering:** The system is based on the general search framework of constraint programming and can be easily changed, extended, maintained and reused.

The architecture represents a significant step toward realization of intelligent agents for computer games. Not only genres like computer role-playing games and strategy games can benefit from this. For games like first-person shooters, too, increasing efforts are being made to include sophisticated behavior for their bots. Of course, the technology is not restricted to computer games. Other application areas include internet agents, factory robots, autonomous spacecraft and many more.

7. Future Work

This book has presented a number of concepts and techniques enabling intelligent agents to be created for a dynamic real-time environment. Apart from research on specialized planning constraints and further empirical application studies, there are, of course, a vast number of issues that still need to be explored and addressed in detail. This chapter identifies some of the issues and offers some ideas on how to tackle them.

7.1 Search Guidance

In many cases, local search methods have problems searching structured search spaces. They repair in an unfocused manner without distinguishing between different problem features. Using global constraints improves the situation, but techniques for coordinating the constraints need to be further explored. An unfocused selection of a repair – e.g., with the steepest descent for an actual inconsistency – need not represent the optimal behavior because a specific region of the search space may require very different improvement heuristics in order to move quickly toward an optimum. Getting trapped in cycles or randomly moving about the search space may be the consequence. For example, the following specification results in a search behavior in which the constraints' actions continually undo one another:

Constraint 1:
 Relation: `y > x + 10`
 Knowledge/Heuristic: `decr x`

Constraint 2:
 Relation: `y > 10 - x`
 Knowledge/Heuristic: `incr x`

Init: `x = 0, y = 0`

There is, then, a need for techniques that perform meta-management, e.g., that **enable local search to temporally focus on specific problem aspects**. This can be supported by the global constraints' domain-specific knowledge.

Some work has already been done on the introduction of focus methods for local search, e.g., modifying the cost function [181, 56, 164], using a special prioritizer [93], making dynamic changes to the neighborhood [13] and employing co-evolution techniques for genetic algorithms [143]. The development of techniques adapted to the use of global constraints would appear a promising line of research, in which domain-knowledge can be used to set and maintain the focus.

7.2 Combination of Local and Refinement Search

The methods used in this book also have their drawbacks. One of these has to do with the use of local search. Local-search techniques are very good at quickly finding high-quality solutions. However, for problems with a very low density of solutions in the search space, systematic approaches employing complete refinement search usually outperform local search. It therefore seems like a good idea to develop hybrid solutions. Work in this area has recently begun, e.g., on enhancing local search with propagation techniques [149, 95] and using refinement search for subproblems within local search [117]. An application to planning is described in [67].

7.3 Learning

The self-adapting preference values used for the Orc Quest example and for the SRC's selection of a heuristic already represent a simple kind of learning. A related approach is presented in [51], local-search methods being applied here to find a subset of heuristics that perform well for a given problem. However, this approach only provides information on which heuristics to apply generally, and we are more interested in approaches that can adapt the distributions during search because the utility of a heuristic probably depends strongly on the current search state.

Another highly interesting topic in this context is the use of case-based reasoning. The cases can either lead to suggestions on which heuristic to apply, or they can directly include concrete improvement transformations, for which the adaptation is automatically performed by local search's repair mechanisms. An overview of case-based reasoning techniques for CSPs is given in [168]. Case-based planning approaches were developed by Hammond [80], Veloso, Muñoz-Avila and Bergmann [178], and Lieber and Napoli [112], among others.

A further important issue is the synthesis of new operators/neighborhoods for planning based on local search. Research on this topic has only just begun, e.g., the work of by Ambite, Knoblock and Minton [8], and will probably intensify as local-search approaches become more widely used for planning.

7.4 Self-Reflective Planning

In general, the planning process itself should also be taken into account by an agent, i.e., the agents being able to plan their own planning. This would enable higher-level matters to be planned, e.g., the organization of optimization and satisfaction phases for the global search control. To solve this task, a special REASONING ARC can be introduced by means of which the reasoning processes are scheduled.

Using the REASONING ARC makes hierarchical planning easy to accomplish as well. Higher-level TASK CONSTRAINTS can involve an additional ACTION TASK for the REASONING ARC, which is temporally located well ahead of the "real" tasks of the TASK CONSTRAINT and causes them to be replaced by more detailed tasks when the ACTION TASK is executed.

7.5 Social Aspects

Little has been said about communication with other agents. A great deal of research on this topic focuses on specific protocols and the like. Our interest in this topic is not only of a syntactical nature, however.

Recently, game-theoretical research topics like *auctions* and *market-oriented programming* have received a lot of attention from the AI community. However, these approaches are rarely applicable for a more general multi-agent system. Some aspects that are usually neglected are:

- Temporal issues:
  - The price of a service is dependent on the time at which the service is implemented and at which the payment is transacted.
  - Prices are dependent on the time spent calculating the prices (e.g., with more time to think about it, easier implementation methods could be found by the service provider).
- Marketplace:
  - There is no central marketplace, and negotiations can involve single agents as well as groups of agents.
  - The search for trade partners and the negotiations themselves involve costs.

Research on these questions has so far been limited, but fields like *bargaining and search* (e.g., see [197]) are starting to attract more attention.

In the context of this book, the only incentive for agents is the prospect of improving their plan quality. That agents will try to obtain help from other agents is nothing special, because it serves their purposes. It is not clear, however, how they can be motivated to help others.

To remedy this situation, two additional *State Resource*s for each familiar agent or group of agents can be introduced: an OWN COOPERATIVENESS SRC resource to express an agent's willingness to help other agents, and an

Other's Cooperativeness SRC resource for another agent's willingness to help one. The level of both resources has an impact on the goal function. The quality of this impact enables the social attitude of an agent to be described.

When an agent receives help from another agent, its Own Cooperativeness SRC is increased, while its Other's Cooperativeness SRC is decreased. For the other agent, the reverse is the case. The level of change is calculated according to the improvement of the first agent's goal function and the worsening of the second agent's goal function.

An agent that plans to be helped by another agent suggests a number of actions to the other agent and specifies the level of potentially gained improvement. If the other agent's increase in Other's Cooperativeness and decrease in Own Cooperativeness results in an improvement in his goal function, he will agree to the cooperation. Once cooperation is confirmed, a failed transaction of the stipulated service would entail a sharp decrease in the Other's Cooperativeness. The communication process could be planned using the Reasoning ARC.

To realize more sophisticated communication, the underlying framework of constraints may turn out to be a major help, because constraints represent a powerful tool for communicating an agent's restrictions and needs.

A. Internet Links

A.1 General AI

– About.com: Artificial Intelligence:
  `http://ai.miningco.com/compute/software/ai`
– American Association for Artificial Intelligence:
  `http://www.aaai.org/`
– CMU Artificial Intelligence Repository:
  `http://www.cs.cmu.edu/afs/cs.cmu.edu/project/` →
  → `ai-repository/ai/html/intro.html`
– Russell and Norvig's AI on the Web:
  `http://www.cs.berkeley.edu/~russell/ai.html`
– Artificial Intelligence Resources of the National Research Council of Canada:
  `http://ai.iit.nrc.ca/ai_point.html`
– Hubat: Artificial Intelligence:
  `http://www.hubat.com/servlets/search?cmd=b&db=hubat&` →
  → `concept=3.1&next=1`
– `comp.ai` Newsgroup FAQs:
  `http://www.faqs.org/faqs/by-newsgroup/comp/comp.ai.html`
– ACM Special Interest Group on Artificial Intelligence:
  `http://sigart.acm.org/`
– The European Coordinating Committee for Artificial Intelligence:
  `http://www.eccai.org/`
– The International Joint Conferences on Artificial Intelligence:
  `http://www.ijcai.org/`
– Principles of Knowledge Representation and Reasoning:
  `http://www.kr.org/`
– The Society for the Study of Artificial Intelligence and the Simulation of Behavior:
  `http://www.cogs.susx.ac.uk/aisb/`

A.2 Artificial Intelligence for Computer Games

– International Game Developers Association's Artificial Intelligence SIG:
  `http://www.igda.org/SIGs/game_ai.htm`

- Steven Woodcock's Game AI Page:
  `http://www.gameai.com/ai.html`
- Craig W. Reynolds's Collection about Improvisational Characters:
  `http://www.red3d.com/cwr/characters.html`
- Gamedev.net – AI Resources:
  `http://www.gamedev.net/reference/list.asp?categoryid=18`
- `comp.ai.games` Newsgroup
- Gamasutra (subscribe to access the Artificial Intelligence in Computer Games discussion group):
  `http://www.gamasutra.com/`

A.3 Agents

- MultiAgent systems:
  `http://www.multiagent.com/`
- UMBC AgentWeb:
  `http://agents.umbc.edu/`
- AgentLink:
  `http://www.agentlink.org/`
- The Agent Society:
  `http://www.agent.org/`
- Patrick Doyle's Notes on Agent Architectures:
  `http://www-cs-students.stanford.edu/~pdoyle/quail/` →
  → `notes/pdoyle/architectures.html`
- Agent Construction Tools:
  `http://www.agentbuilder.com/AgentTools/`
- Jack Krupansky's Software Agent Links:
  `http://www.basetechnology.com/aglinks.htm`
- University of Zürich – AI Lab's Robotics Links:
  `http://www.ifi.unizh.ch/groups/ailab/links/robotic.html`
- Foundation for Intelligent Physical Agents:
  `http://www.fipa.org/`

A.4 Planning

- U.K. Planning and Scheduling Special Interest Group:
  `http://www.research.salford.ac.uk/plansig/`
- Electronic Colloquium on Planning and Scheduling:
  `http://www.informatik.uni-ulm.de/ki/Etai/ec/ec.html`
- Electronic Colloquium on Reasoning about Actions and Change:
  `http://www.ida.liu.se/ext/etai/rac/`
- European Network of Excellence in AI Planning:
  `http://planet.dfki.de/`

- AIAI Planning Resources:
  `http://www.aiai.ed.ac.uk/links/planning.html`
- Rao's Planning Class:
  `http://rakaposhi.eas.asu.edu/planning-class.html`
- Patrick Doyle's Notes on Planning:
  `http://www-cs-students.stanford.edu/~pdoyle/quail/` →
  → `notes/pdoyle/planning.html`
- Rob Miller's Reasoning about Action List:
  `http://www.ucl.ac.uk/~uczcrsm/ReasoningAboutActions.html`
- Rob St. Amant's AI Planning Resources:
  `http://www.csc.ncsu.edu/faculty/stamant/planning-resources.html`
- AIPS-00 Planning Competition:
  `http://www.cs.toronto.edu/aips2000/`
- AI Planning Systems: Domain Repository:
  `http://www.cs.umd.edu/projects/planning/index.html`

A.5 Search Frameworks

- Stas Busygin's NP-Completeness Page:
  `http://www.busygin.dp.ua/npc.html`

A.5.1 Operations Research

- Michael Trick's Operations Research Page:
  `http://mat.gsia.cmu.edu/index.html`
- J E Beasley's OR-Notes:
  `http://mscmga.ms.ic.ac.uk/jeb/or/contents.html`
- `sci.op-research` Newsgroup FAQs:
  `http://www.faqs.org/faqs/by-newsgroup/sci/sci.op-research.html`
- tutOR:
  `http://www.tutor.ms.unimelb.edu.au/`
- Mathematical Programming Glossary:
  `http://www.cudenver.edu/~hgreenbe/glossary/glossary.html`
- NEOS Guide:
  `http://www-fp.mcs.anl.gov/otc/Guide/`
- Interior-Point Methods Online:
  `http://www-unix.mcs.anl.gov/otc/InteriorPoint/`
- The Operational Research Society:
  `http://www.orsoc.org.uk/`
- Decision Tree for Optimization Software:
  `http://plato.la.asu.edu/guide.html`
- INFORMS Online:
  `http://www.informs.org/`

- International Federation of Operational Research Societies:
  `http://www.ifors.org/`
- OpsResearch.com:
  `http://OpsResearch.com/index.html`
- OR-Library:
  `http://graph.ms.ic.ac.uk/info.html`
- Operations Management Resources:
  `http://www.opsmanagement.com/`

A.5.2 Propositional Satisfiability

- SATLIB:
  `http://www.satlib.org/`
- Ian P. Gent and Toby Walsh's References on Satisfiability:
  `http://dream.dai.ed.ac.uk/group/tw/sat/`
- Online MAX-SAT Solver:
  `http://rtm.science.unitn.it/intertools/sat/`

A.5.3 Constraint Programming

- Constraints Archive:
  `http://www.cs.unh.edu/ccc/archive/`
- Roman Barták's On-line Guide to Constraint Programming:
  `http://kti.ms.mff.cuni.cz/~bartak/constraints/`
- Finite Domain Constraint Programming in Oz. A Tutorial:
  `http://www.mozart-oz.org/documentation/fdt/index.html`
- Doug Edmunds' Page on Learning Constraint Logic Programming:
  `http://brownbuffalo.sourceforge.net/`
- The APES Group's Kappa Pages:
  `http://www.dcs.st-and.ac.uk/~apes/kappa.html`
- CSPLib: a Problem Library for Constraints:
  `http://csplib.cs.strath.ac.uk/`
- `comp.constraints` Newsgroup FAQ:
  `http://www.faqs.org/faqs/by-newsgroup/comp/comp.constraints.html`
- ERCIM Working Group on Constraints:
  `http://www.cwi.nl/ERCIM/WG/Constraints/`

B. The "Send More Money" Problem

This example describes the classic crypt-arithmetic puzzle "send more money". A different digit for each letter of the following equation has to be found:

```
  S E N D
+ M O R E
---------
M O N E Y
```

To specify the problem for a constraint solver, we have to declare the **variables** and their **domains**:

```
{S, E, N, D, M, O, R, Y} :: 0..9
```

In addition, the problem involves the following two **constraints**:

```
1)                   1000*S + 100*E + 10*N + D
                   + 1000*M + 100*O + 10*R + E
        = 10000*M + 1000*O + 100*N + 10*E + Y

2)      alldifferent({S, E, N, D, M, O, R, Y})
```

When this information is given to a constraint solver, a solution is produced immediately:

```
{S = 2, E = 8, N = 1, D = 7, M = 0, O = 3, R = 6, Y = 5}
```

Note that no solving method was given. The solver produced only one solution, but all other possible solutions can be listed as well. If special solutions are preferred, an optimization can be applied.

C. Choice Randomization

A lot of decisions that are made during an agent's behavior computation include randomized choices (e.g., see Chap. 2 and 5). This chapter argues in favor of a "fair" random choice and points out that the computation of an alternative's utility should be focused instead of empirically tuning choice parameters of less informed selection methods.

Many researchers do not really care how a randomized choice between several alternatives is made. Of course, in the case of a choice for which no quantitative utility information is available to distinguish the alternatives, the simple random choice of one alternative may be appropriate. However, especially in optimization domains – in which values related to the objective function are often accessible – more sophisticated methods are rarely used. For example, in GSAT [165] or Walksat [166], the choice only depends on which alternative's objective function value is the best. The quantitative utility information is not used.

Bresina introduced a method called heuristic-biased stochastic sampling (HBSS) [24]. The first step of this method is to map the utility values to a ranking of directly consecutive natural numbers, starting with a rank of 1 for the best utility value. But, again, the quantitative utility information is destroyed by this step.

Arguing that the relative differences between the utility values are important, Oddi and Smith developed an approach that uses a dynamic "acceptance band" [142]. A simple random choice is made, considering only alternatives within a certain percentage range from the maximal (if higher values are considered to be better than lower ones) utility value of the alternatives. However, even this approach uses the relative distances only for preselection for a simple random choice.

It is argued below that a fair random choice, in which an alternative is selected with a choice probability proportional to the alternative's utility value, is more appropriate and can be computed faster. It is assumed that the utility function provides values in such a way that a utility value grows proportionally to the preference, e.g., that a utility value of 10 is twice as preferable as a utility value of 5.

C.1 Choice-Dependent Information

The general problem when taking into account the quantitative utility information is that the relative distances between the values decrease if the values increase. For example, let there be two alternatives A and B with utility values of $v_A = 3$ and $v_B = 5$. The superiority of v_B with respect to v_A's value is 66 %. However, if the utility values are $v_A = 1,003$ and $v_B = 1,005$, the superiority of v_B is only about 0.2 %. Is this appropriate or not? This difficulty is perhaps the reason why Bresina chose to make a ranking of the alternatives.

The question here is which reference values should be taken to calculate the percentage. For example, to determine what is 100 % in the above example, the value of 0 was taken as the lower reference value and the value of v_A as the upper reference value.

The key to answering the question of which lower reference value r_l should be taken is to determine which parts of the values are independent of the choice. These independent parts should not affect the choice; they thus represent the lower reference value. For example, if $v_A = 1,003$ and $v_B = 1,005$ and the component that is responsible for a partial value of 1,000 is not dependent on any of the alternatives, r_l should be set to 1,000. On the other hand, if the total values of v_A and v_B are dependent on the choice, r_l should be set to 0.

Consequently, instead of taking the *total* inconsistency that results from choosing an alternative, only the inconsistency *improvement/contribution* of an alternative is taken for most decisions treated in this book. Usually, the utility function that computes the alternatives' utility values should already provide values that include the choice-dependent information only. This is assumed in the following sections (implying that $r_l = 0$).

Choice-dependent utility information also means that the utility of an alternative may be dependent on the other alternatives, e.g., if the sum of the utility values of one category of alternatives is to be equal to the sum of the utility values of another category. Thus, it is often useful to add/incorporate a transformation to the utility function that computes the *relative* utility values.

C.2 Choosing an Alternative

Unlike other approaches, the complete relative differences between the alternatives are preserved here – an approach that is "fairer" and often easier to calculate. An example of the application of such a choice is ant colony optimization [42] in which the probability distribution for an ant's decision (on which way to take) is often made in this way.

When making the choice, every single relevant part of all alternatives has to be considered – without any information-destroying operations like ranking or precluding alternatives. The upper reference value r_u is thus set to the total sum of the alternatives' utility values:

$$r_u = \sum_{a \in \mathcal{A}} v_a$$

This results in the following choice probability for an alternative i:

$$p_i = \frac{v_i}{\sum_{a \in \mathcal{A}} v_a}$$

The implementation of the choice itself is very simple (see Fig. C.1).

```
IF ( r_u > 0 ) BEGIN

   r  ←  random_value([1..r_u])
   a  ←  first_alternative()
   r  ←  r - utility_value(a)

   WHILE ( r > 0 ) BEGIN

      a  ←  next_alternative()
      r  ←  r - utility_value(a)

   END

END
```

Fig. C.1. Fair Random Choice

C.3 Utility Function vs. Random Choice

Determining the utility values is usually a matter of heuristics. The estimate, then, may be sometimes better and sometimes worse. But this is no reason to apply operations like a quadratic function to the utility values (as is done in Bresina's HBSS – after the ranking). The rating done by the utility function is destroyed by this mapping and the relative differences between the alternatives are not preserved. The improvement of the quality of the decisions by applying operations like a quadratic function indicates that the heuristic to determine the utility values should be changed accordingly. Changing the random choice instead of the utility function is like fighting a disease by alleviating its symptoms instead of eliminating its cause.

A good example here is the choice of a modification alternative in the solution to the Orc Quest problem (see Sect. 5.1.2). The solution process is done in a local search way such that in each iteration, there are six alternatives for modifying a solution. These alternatives correspond to adding/removing one action of each type. The utility function is dynamic and assigns a utility according to the improvements that are gained by the application of an alternative. The fair random choice is compared to Bresina's heuristic-biased stochastic sampling and Oddi and Smith's acceptance-band method. The acceptance-band method requires some tuning for the percentage range. The results are given in Fig. C.2 which shows only the graphs for reaching all of the goal criteria.

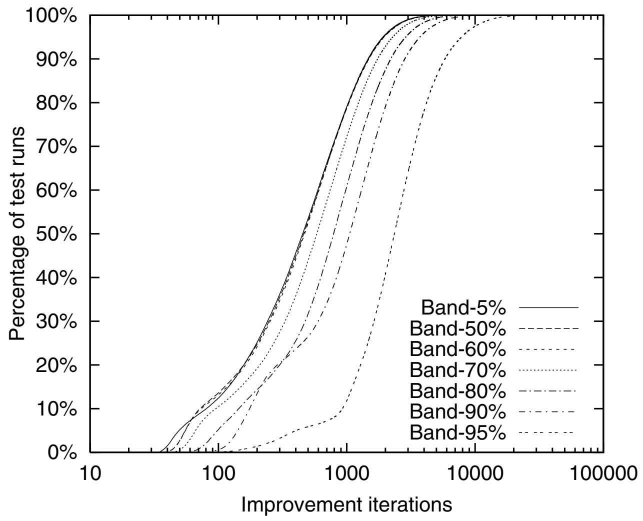


Fig. C.2. Test Runs with Acceptance Bands

It is evident that the quality of the solutions improves for smaller acceptance bands, the best results being actually obtained when there is no acceptance band (a simple random choice between all alternatives with the best utility values). Figure C.3 shows a comparison of all choice techniques, "Best" being a simple random choice between all alternatives with the best utility values, "Rank-Poly-2" an HBSS choice with a bias function of $\frac{1}{r^2}$ (see [24] for details) and "Rank-Poly-3" an HBSS choice with a bias function of $\frac{1}{r^3}$.

The fair random choice is quite competitive, with only "Best" dominating. Higher exponents for the bias function of HBSS have a positive effect, also showing the advantage of choosing the alternative that was rated best. Does this mean that a fair choice is not the best way to proceed?

No! The results do not demonstrate the superiority of one randomization approach over the others, but indicate that the utility function is not set appropriately. The utility function should have been set in such a way that the most successful alternative gets a utility of one and all other alternatives utility values of zero. If the utility function is modified this way, all randomization approaches produce the same results as "Best".

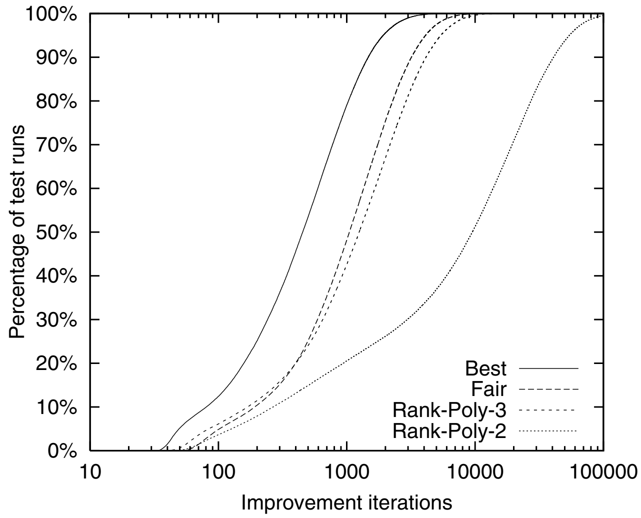


Fig. C.3. Comparison of the Choice Techniques

However, this does not mean that all randomization approaches are the same. By adjusting the utility function, an application of the fair random choice can always simulate all other types of random choice because it can adequately handle quantitative information as well. For example, to realize the "Best" choice, all utilities can be modified as described above; to realize acceptance bands,the utility function must be changed such that all relevant alternatives get a utility value of one; and to realize a ranking, the ranking procedure must be incorporated into the utility function. This ability to realize other randomization approaches cannot be fully exploited in the other approaches because they are not capable of properly considering the quantitative difference information provided by the utility function, e.g., if a selection is to be done with a choice probability proportional to the alternative's utility value.

The fair random choice should therefore be used, and the developer should concentrate on improving the utility function. Using the other techniques focuses the developer on empirically tuning things like a bias function or a percentage for the acceptance bands (the tuning possibilities actually being highly restrictive because the utility information cannot be fully exploited) instead of thinking about better ways to calculate utility. If it turns out, at the end of the algorithm design phase, that the best selection method is only a reduced scheme of the fair random choice – as in the Orc Quest example above – it is, of course, very reasonable to optimize and integrate the choice method with the utility calculation.

C.4 Calculation Costs

For local search techniques, one of the main aspects of a good neighborhood and utility function is that the utility values of the neighbors can be quickly computed. If similar choices are repeatedly made, like choosing one of the six alternatives in the Orc Quest example or choosing a variable to flip its value for SAT problems, usually only a small number of the utility values must be recomputed. Here, the fair random choice is not only more appropriate but also faster to compute. The fair random choice requires only a **sum value** (r_u) instead of the **maximal value** required by Oddi and Smith's acceptance bands or Bresina's ranking. Figure C.4 shows that the results of the fair random choice, compared in terms of time instead of iterations, are much better compared with Fig. C.3.

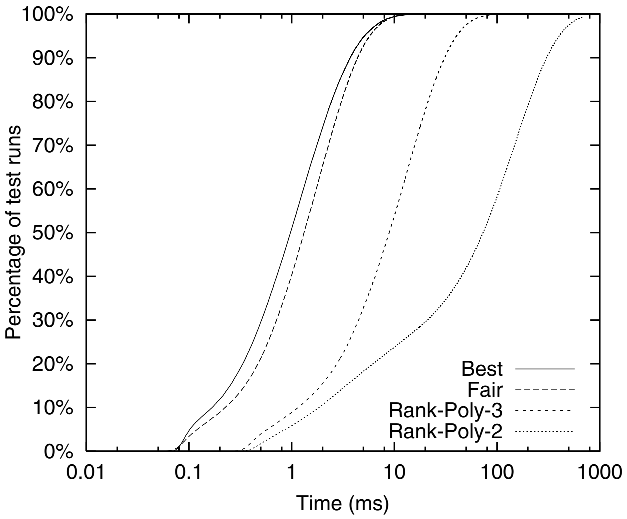


Fig. C.4. Temporal Comparison

In most cases, it is easy to continuously update the fair random choice's sum value instead of recalculating it for each choice. For example, the alternatives that are used for the Orc Quest problem can always update an internal sum value if a utility is changed. This is more efficient than a recalculation because only one utility value is changed between two choices, i.e., one calculation instead of the six that would be necessary to sum up the six utility values. Whether a continuous update or a recalculation is more efficient depends, of course, on the number of changes between two choices, but local search methods are normally constructed in a way such that there is not much change in one search step.

A continuous update of a required maximal utility value is computationally more expensive because in a case in which the maximal utility value must be lowered an update depends on all other values. However, an ordered list can be maintained in which each list element includes a link to one alternative with its utility value. For every value change, the list order is updated according to the utility values. With this list, it is possible to dispense with a recalculation of the maximal value for each choice. But the fair random choice can also be accelerated by this list because consideration of the alternatives in the list's order increases the probability that the choice loop (see Fig. C.1) will be successful sooner. Again, the advantage of the continuously updated list is dependent on the number of changes between two choices. Examples of the use of such lists in this book are the global search control's constraint selection and the selection of an inconsistent interval by an ARC.

C.5 Allowing for Non-Positive Utility Values

Sometimes, a potential alternative actually has a negative utility. But it would seem like a good idea to consider even these alternatives for the choice. The question is, though: in which relation should a choice probability for an alternative with a negative utility value be with respect to the other alternatives? The example considered below includes alternatives A, B and C with utility values $v_A = 2$, $v_B = 5$ and $v_C = -12$, respectively.

Setting all negative values to a value of 1, such that $v_A = 2$, $v_B = 5$ and $v_C = 1$ in our example, is not fair because the relative distances between the values would be lost. A simple shift making the lowest value 1, such that $v_A = 15$, $v_B = 18$ and $v_C = 1$ in our example, is not convincing either because the relative distances between the positive values would decrease with lower negative values (see also Sect. C.1). For example, the resulting probabilities would be 44 % for alternative A, 53 % for alternative B and 3 % for alternative C if v_C is -12, and 49 % for alternative A, 50 % for alternative B and about 0 % for alternative C if v_C were -100.

Additions and subtractions destroy the relative distances between the values and multiplications or divisions cannot produce positive signs for all values. Indeed, the problem here is the lack of an interpretation possibility for negative values.

It is therefore assumed that the interpretation is given by way of a *risk value* r, which provides a probability with which an alternative with a negative utility must be chosen. Another value must be given to provide a probability with which an alternative with a utility value of 0 must be chosen – the *exploration value* e. If there are no utility values to which e or r can be applied, their percentage is added to the one for positive alternatives. This results in the following choice probability for an alternative i:

$$
\begin{aligned}
n_s &= \sum_{a\in\mathcal{A}, v_a>0} v_a \\
e_s &= \sum_{a\in\mathcal{A}, v_a=0} 1 \\
r_s &= \sum_{a\in\mathcal{A}, v_a<0} \frac{1}{v_a}
\end{aligned}
$$

$$
n' = \begin{cases} 1 & : \ n_s > 0 \wedge e_s = 0 \wedge r_s = 0 \\ 1-e & : \ n_s > 0 \wedge e_s > 0 \wedge r_s = 0 \\ 1-r & : \ n_s > 0 \wedge e_s = 0 \wedge r_s < 0 \\ 1-e-r & : \ n_s > 0 \wedge e_s > 0 \wedge r_s < 0 \\ 0 & : \ n_s = 0 \end{cases}
$$

$$
e' = \begin{cases} 1 & : \ n_s = 0 \wedge e_s > 0 \wedge r_s = 0 \\ e & : \ n_s > 0 \wedge e_s > 0 \\ \frac{e}{e+r} & : \ n_s = 0 \wedge e_s > 0 \wedge r_s < 0 \\ 0 & : \ e_s = 0 \end{cases}
$$

$$
r' = \begin{cases} 1 & : \ n_s = 0 \wedge e_s = 0 \wedge r_s < 0 \\ r & : \ n_s > 0 \wedge r_s < 0 \\ \frac{r}{e+r} & : \ n_s = 0 \wedge e_s > 0 \wedge r_s < 0 \\ 0 & : \ e_s = 0 \end{cases}
$$

$$
p_i = \begin{cases} \frac{n' \times v_i}{n_s} & : \ v_i > 0 \\ \frac{e'}{e_s} & : \ v_i = 0 \\ \frac{r'}{v_i \times r_s} & : \ v_i < 0 \end{cases}
$$

For the test runs included in this book, static values of $e = 0.05$ and $r = 0.05$ were used.

C.6 Conclusion

It was argued that it is a bad idea to start the development of a search algorithm with random selection schemes like acceptance bands or rankings. These methods do not appropriately implement the utility function's rating and force the developer to focus on empirically tuning involved parameters instead of thinking about better ways to calculate utility. The fair random choice does not have these drawbacks and can often even be calculated faster. Many examples of using the fair random choice can be found throughout this book.

We do not want to claim, however, that the fair random choice is always superior to other selection schemes. Indeed, it has been shown that this is often not the case. But the fair random choice should be the starting point for developers of a search algorithm, giving them all possible options and focusing them on the development of appropriate neighborhoods and heuristics to determine the utility. Abandoning the fair random choice early on may cause potential solving strategies to be blocked.

D. Ensuring the Satisfaction of Structural Constraints

For smaller applications, it is not necessary to implement a system that supports full structural satisfaction, and – as described in Chap. 5 – this has not been implemented for the test runs described in this book. However, in such cases, it must be proved that the system cannot reach a structurally inconsistent state. This has the drawback that the system is highly inflexible because these proofs have to be redone/rechecked each time the problem specification or the solving mechanisms are changed. Thus, from a software engineering point of view, the better choice is a full implementation of the structural satisfaction (see Sect. 3.7).

The approach of proving the structural consistency for a system a priori does have an advantage, though. It enables the overhead for checking the structural constraints to be dropped during runtime, which results in a speedup for the actual solving process. This chapter describes how to handle the proofs for a system that adopts this approach. It must be guaranteed that the system starts with a structurally consistent constraint graph and that the system's improvement heuristics cannot produce structurally inconsistent graphs. Using the example of the planning model, the following sections show how this can be accomplished.

D.1 A Structurally Consistent Start Graph

In contrast to an approach that features full structural satisfaction, the system cannot use the empty graph as the start graph because the goal specification's structural constraint S_{Goals} (see Sect. 4.2.10) would be inconsistent.

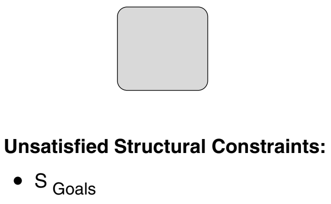


Fig. D.1. Creating the Start Graph: Initial Phase

The constraint S_{Goals} is the only structural constraint prevents the empty graph from being used. Thus, in the following, the empty graph will be extended in a such way that the constraint S_{Goals} is satisfied. For this extension, the productions that are derived from the search-space generation process of Sect. 3.5 must be used because these implicitly ensure that the SCSP's embedding and extension graphs are not violated, which is not guaranteed by the planning SCSP's structural constraints $\mathcal{S}$.

The following generation process for the start graph does not create any redundant structures. The structural constraints that are introduced by the methods described in Sect. 3.6 to prevent redundancies do therefore not have to be tested.

D.1.1 The Start Graph's Variables and Equals:X Constraints

To begin with, the variables of S_{Goals}'s testing part can be created by applying the production P_{v_a} (see Fig. D.2 for an example using S_{Goals} of Fig. 5.2). This cannot entail further inconsistent structural constraints as there is no structural constraint that docks at variables only.

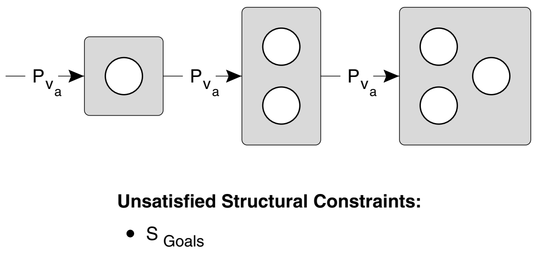


Fig. D.2. Creating the Start Graph: Phase 1

The start graph's constraints that restrict the variables to specific values (a constant) are created by applying one distinctive production $P_{Equals:X_a}$ per variable of the graph (see Fig. D.3). The NACs of the productions (see Sect. 3.5.2) are satisfied because no constraints were connected to the variables yet and every application of a production uses a different variable. The addition of the Equals:X constraints cannot entail another inconsistent structural constraint as there is no structural constraint that docks at these constraints.

D.1.2 The Start Graph's State Resources

The State Resources of the constraint S_{Goals} can be included by the production $P_{StateResource_a}$. However, the left-hand side of the production requires the existence of a Current State and a State Projection. These must be created beforehand.

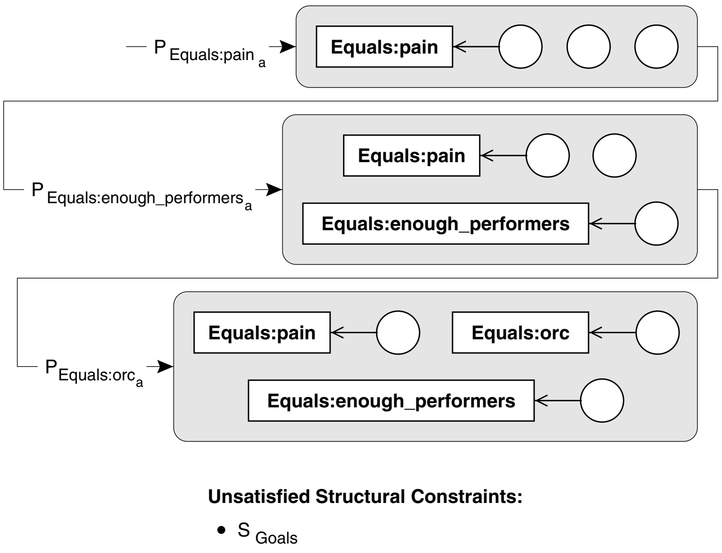


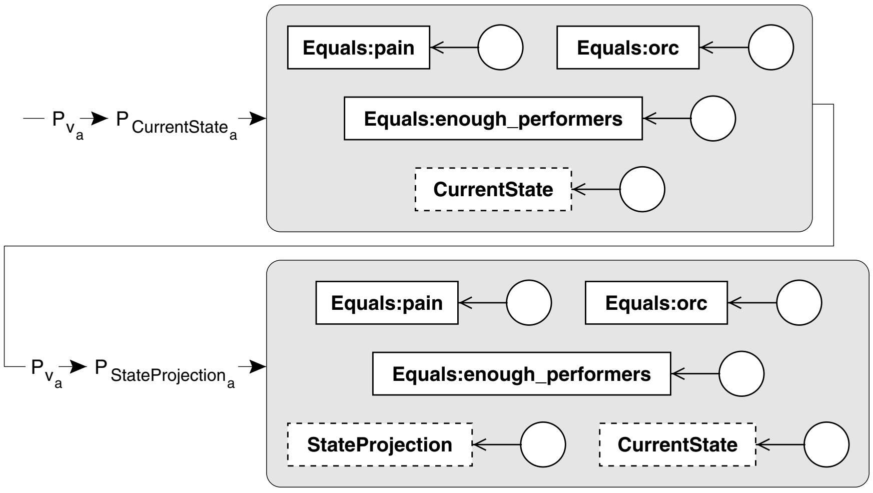
Fig. D.3. Creating the Start Graph: Phase 2

For every State Resource of the constraint S_{Goals}, a Current State is added by the production $P_{CurrentState_a}$. For every application of the production $P_{CurrentState_a}$, the variable that is used within this production is created by the production P_{v_a} beforehand (see Fig. D.4). The structural constraint $S_{CurrentState}$ (see Fig. 4.23) docks at the Current Time and requires the connection with a State Resource. The same procedure/constraints apply for the addition of State Projections.

For every State Resource, a Current State and a State Projection are now available, and the State Resources are created by the production $P_{StateResource_a}$. The variable that is linked to the corresponding Equals:X constraint is used as *ResourceType* variable (see Fig. D.5). This means that the structural constraints $S_{CurrentState}$ and $S_{StateProjection}$ become satisfied. The NACs of the structural constraints' testing parts cannot match the graph because every Current State / State Projection was connected to a different State Resource.

The structural constraint $S_{StateResource}$ can dock at the State Resource (see Fig. 4.26) and requires the linking of an Object to a State Resource Constraint as well as that of the *ResourceType* variable to an All Different constraint.

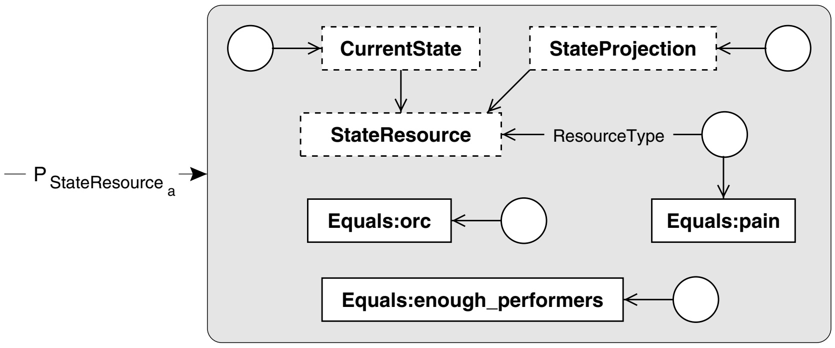
An All Different constraint is created by the production $P_{AllDifferent_a}$, the constraint being connected to one of the *ResourceType* variables (see Fig. D.6). All other *ResourceType* variables are connected to the All Different constraint by production $P_{AllDifferent_e}$. The existence of the All Different constraint entails the applicability of the structural constraint $S_{AllDifferent}$ (see Fig. 4.30). This structural constraint is satisfied because



Unsatisfied Structural Constraints:

- S_{Goals}
- $S_{CurrentState}$
- $S_{StateProjection}$

Fig. D.4. Creating the Start Graph: Phase 3



Unsatisfied Structural Constraints:

- S_{Goals}
- $S_{StateResource}$

Fig. D.5. Creating the Start Graph: Phase 4

its second test alternative can match the graph. The NAC of the second test alternative does not cause any problems because there is no other ALL DIFFERENT constraint that is connected to a STATE RESOURCE's *ResourceType* variable.

For every STATE RESOURCE, one STATE RESOURCE CONSTRAINT must be created. However, the production $P_{StateResourceConstraint_a}$ needed to do this requires the existence of a CURRENT TIME constraint. This is added by the production $P_{CurrentTime_a}$, for which a new variable is created beforehand. The structural constraint $S_{CurrentTime}$ (see Fig. 4.8) is not violated as only one CURRENT TIME constraint was created.

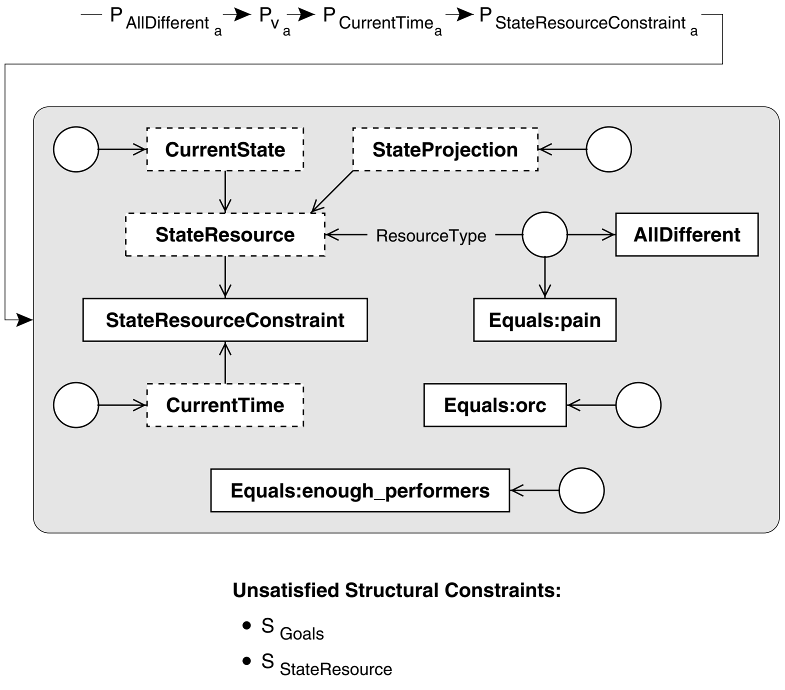


Fig. D.6. Creating the Start Graph: Phase 5

The last thing missing to satisfy the structural constraint $S_{StateResource}$ is the linkage of OBJECTs to the STATE RESOURCEs. In preparation for this, OBJECT constraints must be created[1] by the production P_{Object_a}, which requires the existence of an EXISTENCE CONSTRAINT. Thus, the production

[1] The safest way to do this is to create one OBJECT per STATE RESOURCE. But the developer can use his background knowledge to determine the required number of OBJECTs and to assign the STATE RESOURCEs to them. In the case of wrong or suboptimal assignments, multiple re-assignments and additions/deletions of OBJECTs may be necessary during the search process. To assign multiple STATE

$P_{ExistenceConstraint_a}$ must be applied beforehand. The structural constraint $S_{ExistenceConstraint}$ docks at the EXISTENCE CONSTRAINT (see Fig. 4.36) but is satisfied because only one EXISTENCE CONSTRAINT was produced.

A new variable is created for every OBJECT to be used for the following applications of the production P_{Object_a} (see Fig. D.7). The structural constraint S_{Object} (see Fig. 4.29) can dock at the newly created OBJECT and becomes unsatisfied because of the nonexistent *ObjectType* variable.

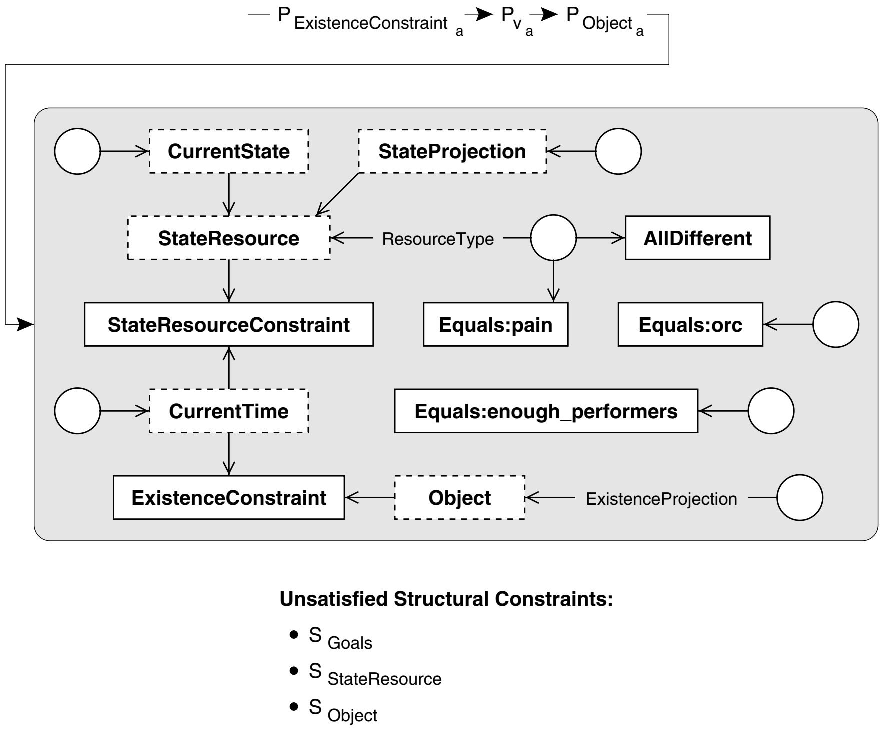


Fig. D.7. Creating the Start Graph: Phase 6

This is fixed by producing a variable for every OBJECT, connecting the variable by the production $P_{Object_{e1}}$ to the OBJECT, and connecting the variables (the first one by the production $P_{AllDifferent_a}$, the others by the production $P_{AllDifferent_e}$) to an ALL DIFFERENT constraint (see Fig. D.8). The existence of the ALL DIFFERENT constraint entails the applicability of the structural constraint $S_{AllDifferent}$ (see Fig. 4.30). This structural constraint is satisfied because its third test alternative can match the graph. The NAC of the third test alternative cannot match the graph as there is only

RESOURCEs to one OBJECT, the structural constraint S_{Object} must be taken into account.

one ALL DIFFERENT constraint that is connected to an OBJECT's *ObjectType* variable.

The structural constraint S_{Object} is now satisfied, and its two NACs cannot endanger the matching of the testing part. The first (upper) NAC is satisfied because there was exactly one *ObjectType* variable produced for and connected to each OBJECT. The second NAC is satisfied as there is currently no STATE RESOURCE connected to the OBJECT.

Finally, every STATE RESOURCE is connected to an OBJECT by the production $P_{Object_{e2}}$. To make it impossible for the second NAC of the structural constraint S_{Object} to match, two STATE RESOURCEs that are linked to the same *ResourceType* variable are not allowed to be connected to the same OBJECT.

As every STATE RESOURCE is connected to an OBJECT, the testing part of the structural constraint $S_{StateResource}$ can match the structures and becomes satisfied. The NAC of the structural constraint cannot match because every STATE RESOURCE is connected to only one OBJECT.

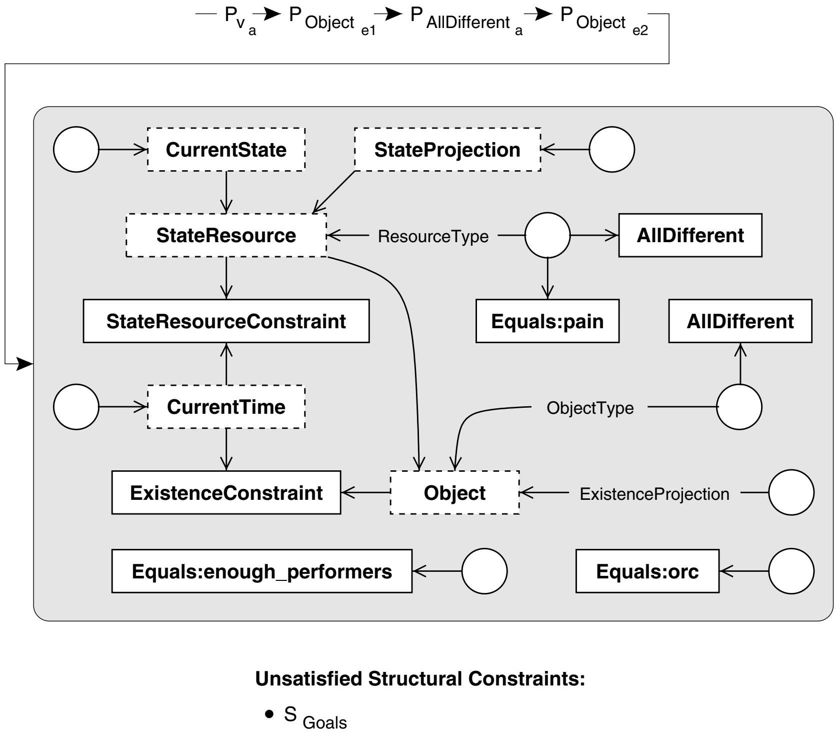


Fig. D.8. Creating the Start Graph: Phase 7

D.1.3 The Start Graph's Task Constraints

For every Task Constraint of the structural constraint S_{Goals}, the production $P_{TaskConstraint_a}$ is applied, using the variable that is linked to the corresponding Equals:X constraint as *ActionType* variable (see Fig. D.9).

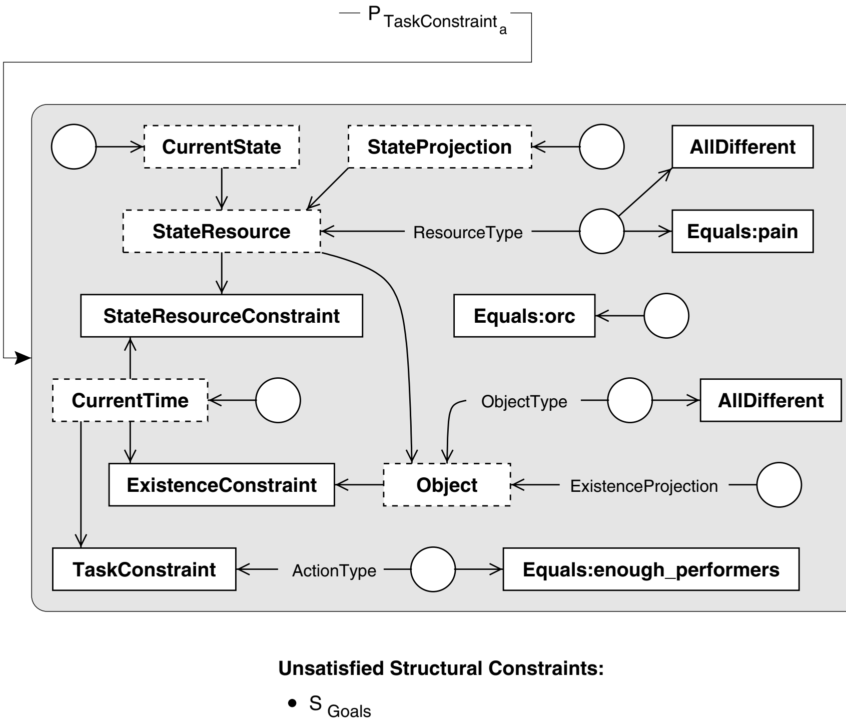


Fig. D.9. Creating the Start Graph: Phase 8

D.1.4 The Start Graph's Action Resource Constraints

The Action Resource Constraints of S_{Goals} are created by the production $P_{ActionResourceConstraint_a}$, using the variable that is linked to the corresponding Equals:X constraint as *ResourceType* variable (see Fig. D.10). The structural constraint $S_{ActionResourceConstraint}$ is unsatisfied by the resulting graph because the *ResourceType* variables are not connected to an All Different constraint (see Fig. 4.15).

An All Different constraint is created by the production $P_{AllDifferent_a}$, the constraint being connected to one of the ARCs' *ResourceType* variables. All other ARCs' *ResourceType* variables are connected to the All Different constraint by production $P_{AllDifferent_e}$. The existence of the All

DIFFERENT constraint entails the applicability of the structural constraint $S_{AllDifferent}$ (see Fig. 4.30). This structural constraint is satisfied because its first test alternative can match the graph. The NAC of the first test alternative does not cause any problems because there is no other ALL DIFFERENT constraint that is connected to an ARC's *ResourceType* variable. The structural constraint $S_{ActionResourceConstraint}$ becomes satisfied by the inclusion of the ALL DIFFERENT constraint. The NAC of the structural constraint cannot match because it is not allowed to have multiple ARCs connected to the same *ResourceType* variable. Otherwise, the S_{Goals} constraint would have been specified in an inconsistent way.

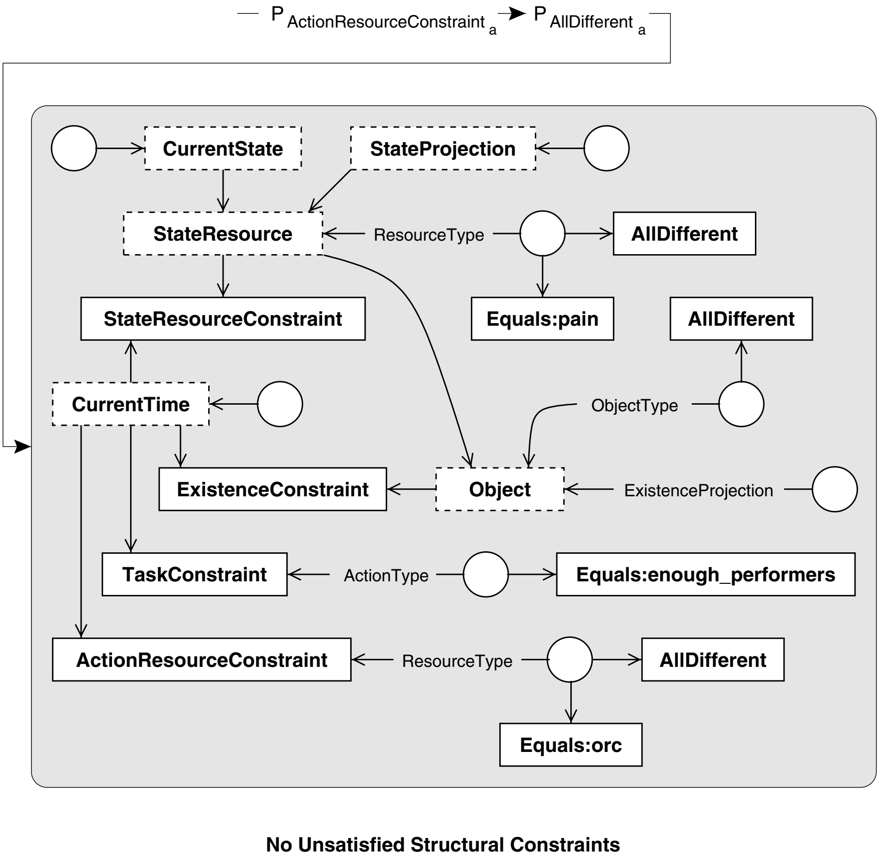


Fig. D.10. Creating the Start Graph: Phase 9

The structural constraint S_{Goals} is now satisfied, and the whole start graph becomes structurally consistent.

D.1.5 Preparation for Closed Worlds

However, for planning problems that make use of the closed world assumption, it is useful to create additionally all involved STATE RESOURCEs and ACTION RESOURCE CONSTRAINTs in advance, so that the improvement heuristics can directly include actions without having to worry about the existence of the required STATE RESOURCEs and ACTION RESOURCE CONSTRAINTs. Of course, this part of the start-graph generation is not mandatory.

To realize this extension, all STATE RESOURCEs and ACTION RESOURCE CONSTRAINTs of the problem that are not included in the structural constraint S_{Goals} are generated in the same way as the other resources beforehand. Phase 2 (see Fig. D.3) is skipped for these additions.

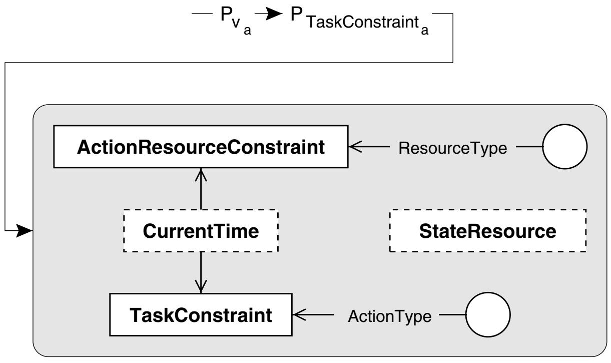
D.2 Validating the Improvement Heuristics

Besides the need to create a structurally consistent start graph, it must be ensured that the improvement heuristics cannot produce a structurally inconsistent graph. A proof for this is highly domain-dependent. It is demonstrated below using the Orc Quest example's heuristics (see Sect. 5.1.2). These heuristics add and remove actions. The Orc Quest example being a closed-world domain, the start graph is created with all possible STATE RESOURCEs and ACTION RESOURCE CONSTRAINTs (i.e., including the process described in Sect. D.1.5).

D.2.1 Adding an Action

If an action is to be added, the first step depends on whether an action of the same *ActionType* already exists. If an *ActionType* variable with the required value does not exist, a new *ActionType* variable is created, to be used by the production $P_{TaskConstraint_a}$ to add the TASK CONSTRAINT (see Fig. D.11; for clarity's sake, only the CURRENT TIME, a STATE RESOURCE and an ACTION RESOURCE CONSTRAINT with its *ResourceType* variable are shown from the current graph). Otherwise, the corresponding *ActionType* variable that already exists is used by the production.

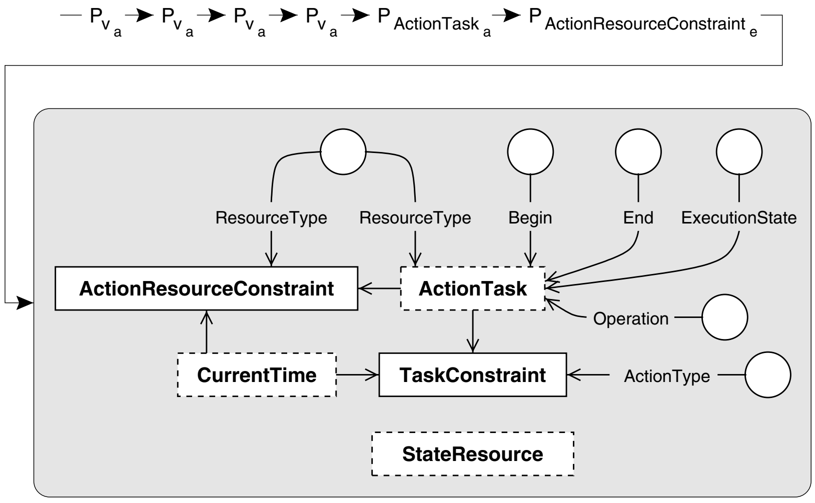
For all ACTION TASKs required by the TASK CONSTRAINT (only one in the Orc Quest example), four variables are created, followed by the application of production $P_{ActionTask_a}$, which uses the ARC's *ResourceType* variable as its *ResourceType* variable (see Fig. D.12). The structural constraint $S_{ActionTask}$ (see Fig. 4.13) docks at the new ACTION TASKs and is unsatisfied because of the missing links to ARCs. This inconsistency is resolved by applications of the production $P_{ActionResourceConstraint_e}$, the ACTION TASKs being linked to the corresponding ARCs (to the ORC ARC).



No Unsatisfied Structural Constraints

Fig. D.11. Validating the Action Insertion: Phase 1

Furthermore, the produced structure entails the applicability of the structural constraint $S_{TaskConstraint}$ (see Fig. 4.10). Since, however, every Action Task was connected to only one Task Constraint, the NAC of the testing part cannot match and the structural constraint is satisfied.



No Unsatisfied Structural Constraints

Fig. D.12. Validating the Action Insertion: Phase 2

For all State Tasks required by the Task Constraint (two in the Orc Quest example), two variables are created, followed by the application of production $P_{StateTask_a}$ (see Fig. D.13). The structural constraint

$S_{DependencyEffect}$ (see Fig. 4.27) docks at the new STATE TASKs and is unsatisfied because of the missing links to TASK CONSTRAINTs or STATE RESOURCE CONSTRAINTs. The production $P_{StateTask_e}$ is applied to all new STATE TASKs to link them to the new TASK CONSTRAINT, which causes the first test alternative of the structural constraint to become satisfied. The NAC does not apply because the STATE TASKs were not connected to STATE RESOURCE CONSTRAINTs

The produced structure entails the applicability of the structural constraint $S_{TaskConstraint}$ (see Fig. 4.10). Since, however, every STATE TASK was connected to only one TASK CONSTRAINT, the NAC of the testing part cannot match and the structural constraint is satisfied.

The structural constraint $S_{StateTask}$ (see Fig. 4.20) docks at the new STATE TASKs as well. It is unsatisfied because of the missing links to STATE RESOURCEs. This inconsistency is resolved by applying the production $P_{StateResource_{e2}}$ for every new STATE TASK, the STATE TASKs being linked to the corresponding STATE RESOURCEs (one to the PAIN STATE RESOURCE, the other to the PERFORMERS STATE RESOURCE). The NAC of $S_{StateTask}$ cannot match as every STATE TASK was connected to only one STATE RESOURCE.

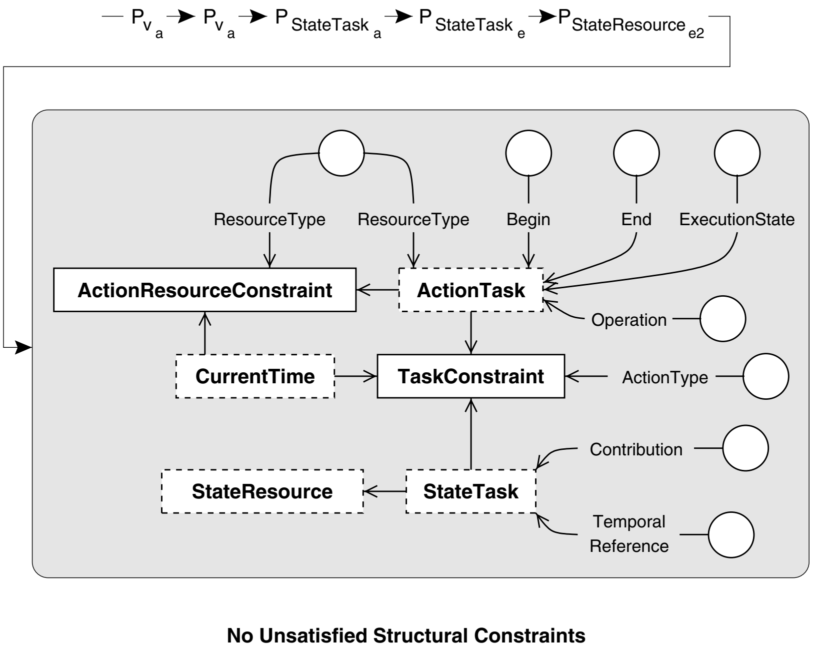


Fig. D.13. Validating the Action Insertion: Phase 3

The actions used by the improvement heuristics in the Orc Quest example do not include any PRECONDITION TASKs – unlike the TASK CONSTRAINT's fourth configuration that is requested by S_{Goals}. Since this extension is needed only once at the beginning, the start graph is directly extended by this PRECONDITION TASK.

For the PRECONDITION TASK, two variables are created, followed by the application of production $P_{PreconditionTask_a}$ (see Fig. D.14; for clarity's sake, only the PERFORMERS STATE RESOURCE and the TASK CONSTRAINT with its *ActionType* variable and the connected EQUALS:ENOUGH_PERFORMERS constraint are shown from the start graph). The constraint $S_{PreconditionTask}$ (see Fig. 4.19) docks at the new PRECONDITION TASK. It is unsatisfied because of the missing link to a STATE RESOURCE. This inconsistency is resolved by applying the production $P_{StateResource_{e1}}$ for the new PRECONDITION TASK, the PRECONDITION TASK being linked to the PERFORMERS STATE RESOURCE. The NAC of $S_{PreconditionTask}$ cannot match because the PRECONDITION TASK was connected to only one STATE RESOURCE.

The produced structure entails the applicability of the structural constraint $S_{TaskConstraint}$ (see Fig. 4.10). Since, however, the PRECONDITION TASK was connected to only one TASK CONSTRAINT, the NAC of the testing part cannot match and the structural constraint is satisfied.

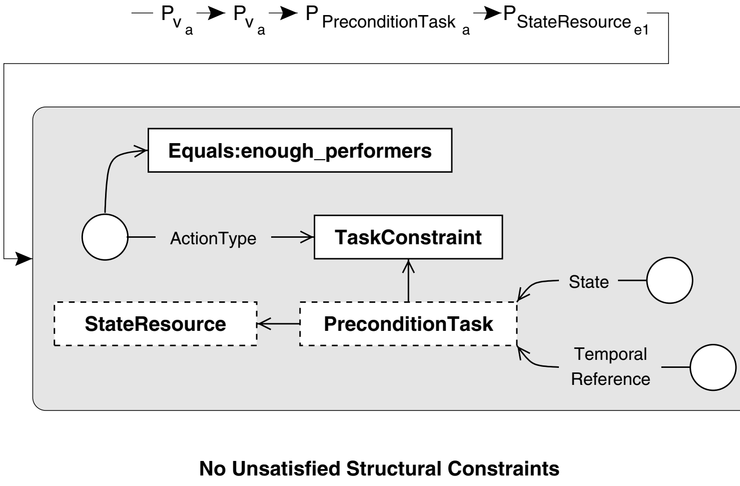

$P_{v_a} \rightarrow P_{v_a} \rightarrow P_{PreconditionTask_a} \rightarrow P_{StateResource_{e1}}$
Equals:enough_performers
ActionType
TaskConstraint
State
StateResource
PreconditionTask
Temporal Reference
No Unsatisfied Structural Constraints

Fig. D.14. Creating the Start Graph: Extension for the Orc Quest Example

D.2.2 Removing an Action

The removal of an action does not need to be considered here as it is simply the reversal of the previous section's productions. Requirement req_p (see Sect. 3.5) guarantees that this is possible.

References

1. Aarts, E. H. L., and Lenstra, J. K. eds. 1997. *Local search in Combinatorial Optimization.* Reading, Wiley-Interscience.
2. Agre, P., and Chapman, D. 1987. PENGI: An Implementation of a Theory of Activity. In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), 268–272.
3. Allen, J. F. 1983. Maintaining Knowledge about Temporal Intervals. *Communications of the ACM* 26(11): 832–843.
4. Allen, J. F. 1984. Towards a General Theory of Action and Time. *Artificial Intelligence* 23: 123–154.
5. Allen, J. F., and Ferguson, G. 1994. Actions and Events in Interval Temporal Logic. *Journal of Logic and Computation* 4(5): 531–579.
6. Allis, L. V.; Van den Herik, H. J.; and Huntjens, M. P. H. 1993. Go-Moku Solved by New Search Techniques. In Proceedings of the 1993 AAAI Fall Symposium on Games: Planning and Learning.
7. Ambite, J. L., and Knoblock, C. A. 1997. Planning by Rewriting: Efficiently Generating High-Quality Plans. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 706–713.
8. Ambite, J. L.; Knoblock, C. A.; and Minton, S. 2000. Learning Plan Rewriting Rules. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000).
9. Ambros-Ingerson, J. A., and Steel, S. 1988. Integrating Planning, Execution and Monitoring. In Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI-88), 83–88.
10. Babaian, T., and Schmolze, J. G. 1999. PSIPLAN: Planning with ψ-forms over Partially Closed Worlds. In Proceedings of the Fifth European Conference on Planning (ECP'99).
11. Bacchus, F., and Kabanza, F. 2000. Using Temporal Logics to Express Search Control Knowledge for Planning. *Artificial Intelligence* 116: 123–191.
12. Baptiste, P., and Le Pape, C. 1995. A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), 600–606.
13. Barbulescu, L.; Watson, J.-P.; and Whitley, L. D. 2000. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 879–884.
14. Barto, A. G.; Bradtke, S. J.; and Singh, S. P. 1995. Learning to Act using Real-Time Dynamic Programming. *Artificial Intelligence* 72(1): 81–138.
15. Bessière, C.; Freuder, E. C.; and Régin, J.-Ch. 1995. Using Inference to Reduce Arc Consistency Computation. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), 592–598.

16. Blum, A. L., and Furst, M. L. 1997. Fast Planning Through Planning Graph Analysis. *Artificial Intelligence* 90: 281–300.
17. Bockmayr, A., and Kasper, T. 1998. Branch-and-Infer: A Unifying Framework for Integer and Finite Domain Constraint Programming. *INFORMS Journal on Computing* 10(3): 287–300.
18. Bockmayr, A., and Dimopoulos, Y. 1998. Mixed Integer Programming Models for Planning Problems. In Working Notes of the CP98 Workshop on Constraint Problem Reformulation.
19. Bonasso, R. P.; Firby, R. J.; Gat, E.; Kortenkamp, D.; Miller, D. P.; and Slack, M. G. 1997. Experiences with an Architecture for Intelligent, Reactive Agents. *Journal of Experimental and Theoretical Artificial Intelligence* 9(1).
20. Bonet, B.; Loerincs, G.; and Geffner, H. 1997. A Robust and Fast Action Selection Mechanism for Planning. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 714–719.
21. Boutilier, C., and Brafman, R. I. 1997. Planning with Concurrent Interacting Actions. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 720–726.
22. Boutilier, C.; Dean, T.; and Hanks, S. 1999. Decision-Theoretic Planning: Structural Assumptions and Computational Leverage. *Journal of Artificial Intelligence Research* 11: 1–94.
23. Brafman, R. I., and Hoos, H. H. 1999. To Encode or not to Encode – I: Linear Planning. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), 988–993.
24. Bresina, J. L. 1996. Heuristic-Biased Stochastic Sampling. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 271–278.
25. Brooks, R. A. 1986. A Robust Layered Control System for a Mobile Robot. *IEEE Journal of Robotics and Automation RA-2* (1): 14–23.
26. Carlier, J., and Pinson, E. 1990. A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem. *Annals of Operations Research* 26: 269–287.
27. Caseau, Y., and Laburthe, F. 1994. Improved CLP Scheduling with Task Intervals. In Proceedings of the Eleventh International Conference on Logic Programming (ICLP'94), 369–383.
28. Chapman, D. 1987. Planning for Conjunctive Goals. *Artificial Intelligence* 32(3): 333–377.
29. Charla, C. 1995. Mind Games: the Rise and Rise of Artificial Intelligence. *Next Generation* 11/95.
30. Chien, S.; Knight, R.; Stechert, A.; Sherwood, R.; and Rabideau, G. 2000. Using Iterative Repair to Improve the Responsiveness of Planning and Scheduling. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000).
31. Chien, S.; Rabideau, G.; Knight, R.; Sherwood, R.; Engelhardt, B.; Mutz, D.; Estlin, T.; Smith, B.; Fisher, F.; Barrett, T.; Stebbins, G.; and Tran, D. 2000. ASPEN – Automating Space Mission Operations using Automated Planning and Scheduling. In Proceedings of the Sixth International Symposium on Technical Interchange for Space Mission Operations and Ground Data Systems (SpaceOps 2000).
32. Coco, D. 1997. Creating Intelligent Creatures. *Computer Graphics World*, July 1997.
33. Coradeschi, S., and Saffiotti, A. 2000. Anchoring Symbols to Sensor Data: preliminary report. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 129–135.

34. Danzig, G. B. 1963. *Linear Programming and Extensions.* Princeton University Press.
35. Davenport, A.; Tsang, E.; Wang, C. W.; and Zhu, K. 1994. GENET: A Connectionist Architecture for Solving Constraint Satisfaction Problems by Iterative Improvement. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 325–330.
36. Davis, M., and Putnam, H. 1960. A Computation Procedure for Quantification Theory. *Journal of the ACM* 7(3): 201–215.
37. Dean, T.; Kaelbling, L. P.; Kirman, J.; and Nicholson, A. 1995. Planning under Time Constraints in Stochastic Domains. *Artificial Intelligence* 76: 35–74.
38. Dechter, R. 1990. Enhancement Schemes for Constraint Processing: Backjumping, Learning, and Cutset Decomposition. *Artificial Intelligence* 41: 273–312.
39. Dechter, R.; Meiri, I.; and Pearl, J. 1991. Temporal Constraint Networks. *Artificial Intelligence* 49: 61–95.
40. Dijkstra, E. W. 1959. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1: 269–271.
41. Do, B., and Kambhampati, S. 2000. Solving Planning Graph by Compiling it into a CSP. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000).
42. Dorigo, M.; Di Caro, G.; and Gambardella, L. M. 1999. Ant Algorithms for Discrete Optimization. *Artificial Life* 5(3), 137–172.
43. Drabble, B. 1993. Excalibur: A Program for Planning and Reasoning with Processes. *Artificial Intelligence* 62(1): 1–40.
44. Drabble, B. and Tate, A. 1994. The Use of Optimistic and Pessimistic Resource Profiles to Inform Search in an Activity Based Planner. In Proceedings of the Second International Conference on AI Planning Systems (AIPS-94), 243–248.
45. Drabble, B.; Dalton, J.; and Tate, A. 1997. Repairing Plans on the Fly. In Proceedings of the 1997 NASA Workshop on Planning and Scheduling for Space.
46. d'Inverno, M.; Kinny, D.; Luck, M.; and Wooldridge, M. 1997. A formal specification of dMARS. Technical Report 72, Australian Artificial Intelligence Institute, Melbourne, Australia.
47. Draper, D.; Hanks, S.; and Weld, D. 1994. Probabilistic Planning with Information Gathering and Contingent Execution. In Proceedings of the Second International Conference on AI Planning Systems (AIPS-94), 31–36.
48. Drummond, M. E., and Bresina, J. L. 1990. Anytime Synthetic Projection: Maximizing the Probability of Goal Satisfaction. In Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90), 138–144.
49. Ehrig, H.; Pfender, M.; and Schneider, H. J. 1973. Graph Grammars: An Algebraic Approach. In Proceedings of the Fourteenth Annual Symposium on Switching and Automata Theory (SWAT), 167–180.
50. El-Kholy, A., and Richards, B. 1996. Temporal and Resource Reasoning in Planning: the *parc*PLAN approach. In Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI-96), 614–618.
51. Engelhardt, B., and Chien, S. 2000. An Empirical Analysis of Local Search in Stochastic Optimization for Planner Strategy Selection. In Workshop Notes of the ECAI-2000 Workshop on Local Search for Planning & Scheduling, 10–16.
52. Ephrati, E.; Pollack, M. E.; and Milshtein, M. 1996. A Cost-Directed Planner: Preliminary Report. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 1223–1228.

53. Erol, K.; Nau, D. S.; and Subrahmanian, V. S. 1991. Complexity, Decidability and Undecidability Results for Domain-Independent Planning. Technical Report CS-TR-2797, University of Maryland, Institute for Advanced Computer Studies, Maryland, USA.
54. Etzioni, O.; Hanks, S.; Weld, D.; Draper, D.; Lesh, N.; and Williamson, M. 1992. An Approach to Planning with Incomplete Information. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92).
55. Fikes, R. E., and Nilsson, N. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. *Artificial Intelligence* 5(2): 189–208.
56. Frank, J. 1997. Learning Short-Term Weights for GSAT. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97), 384–391.
57. Freksa, C. 1992. Temporal Reasoning Based on Semi-Intervals. *Artificial Intelligence* 54: 199–227.
58. Freuder, E. C., and Wallace, R. J. 1992. Partial Constraint Satisfaction. *Artificial Intelligence* 58: 21–70.
59. Funge, J.; Tu, X; and Terzopoulos, D. 1999. Cognitive Modeling: Knowledge, Reasoning and Planning for Intelligent Characters. In Proceedings of the International Conference on Computer Graphics and Interactive Techniques (SIGGRAPH'99), 29–38.
60. Gamma, E.; Helm, R.; Johnson, R.; and Vlissides, J. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Addison-Wesley Professional Computing Series.
61. Gard, T. 2000. Building Character. *Gamasutra*, June 2000. `http://www.gamasutra.com/features/20000720/gard_01.htm`
62. Gasser, R. 1996. Solving Nine Men's Morris. *Computational Intelligence* 12(1): 24–41.
63. Gelfond, M., and Lifschitz, V. 1992. Representing Actions in Extended Logic Programming. In Proceedings of the Joint International Conference and Symposium on Logic Programming (JICSLP'92), 559–573.
64. Gent, I. P.; MacIntyre, E.; Prosser, P.; and Walsh, T. 1997. The Scaling of Search Cost. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 315–320.
65. Gent, I. P., and Walsh, T. 1993. Towards an Understanding of Hill-climbing Procedures for SAT. In Proceedings of Eleventh National Conference on Artificial Intelligence (AAAI-93), 28–33.
66. Georgeff, M. P., and Lansky, A. L. 1987. Reactive Reasoning and Planning. In Proceedings of the Sixth National Conference on Artificial Intelligence (AAAI-87), 677–682.
67. Gerevini, A., and Serina, I. 1999. Fast Planning through Greedy Action Graphs. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 503–510.
68. Glover, F. 1989. Tabu Search – Part I. *ORSA Journal on Computing* 1(3): 190–206.
69. Goldberg, D. E. 1989. *Genetic Algorithms in Search, Optimization, and Machine Learning.* Addison-Wesley.
70. Golden, K., Etzioni, O., and Weld, D. 1994. Omnipotence Without Omniscience: Efficient Sensor Management for Planning. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 1048–1054.

71. Goldman, R. P., and Boddy, M. S. 1994. Conditional Linear Planning. In Proceedings of the Second International Conference on AI Planning Systems (AIPS-94), 80–85.
72. Goldman, R. P., Haigh, K. Z.; Musliner, D. J.; and Pelican, M. 2000. MACBeth: A Multi-Agent Constraint-Based Planner. In Papers from the AAAI-2000 Workshop on Constraints and AI Planning, Technical Report, WS-00-02, 11–17. AAAI Press, Menlo Park, California.
73. Goltz, H.-J. 1995. Reducing Domains for Search in CLP(FD) and Its Application to Job-Shop Scheduling. In Proceedings of the First International Conference on Principles and Practice of Constraint Programming (CP95), 549–562.
74. Goltz, H.-J. 1997. Redundante Constraints und Heuristiken zum effizienten Lösen von Problemen der Ablaufplanung mit CHIP. In Proceedings of the 12. Workshop on Logic Programming (WLP'97), Forschungsbericht PMS-FB-1997-10, LMU München, Germany.
75. Gomes, C.; Selman, B.; and Kautz, H. 1998. Boosting Combinatorial Search Through Randomization. In Proceedings of the Fifteenth National Conference on Artificial Intelligence (AAAI-98), 431–437.
76. Gomes, C. P.; Selman, B.; Crato, N.; Kautz, H. A. 2000. Heavy-Tailed Phenomena in Satisfiability and Constraint Satisfaction Problems. *Journal of Automated Reasoning* 24(1/2): 67–100.
77. Grand, S.; Cliff, D.; and Malhotra, A. 1997. Creatures: Artificial Life Autonomous Software Agents for Home Entertainment. In Proceedings of the First International Conference on Autonomous Agents (Agents'97), 22–29.
78. Gu, J. 1992. Efficient Local Search for Very Large-Scale Satisfiability Problems. *SIGART Bulletin* 3(1): 8–12.
79. Habel, A.; Heckel, R.; and Taentzer, G. 1996. Graph Grammars with Negative Application Conditions. *Fundamenta Informaticae*, Vol. 26, No. 3 & 4.
80. Hammond, K. J. 1990. Case-Based Planning: A Framework for Planning from Experience. *The Journal of Cognitive Science*, 14(3): 385–443.
81. Han, C., and Lee, C. 1988. Comments on Mohr and Henderson's Path Consistency Algorithm. *Artificial Intelligence* 36, 125–130.
82. Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics* 4(2): 100–107.
83. Harvey, W. D., and Ginsberg, M. L. 1995. Limited Discrepancy Search. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), 607–615.
84. Hause, K. 1999. What to Play Next: Gaming Forecast, 1999-2003. Report #W21056, International Data Corporation, Framingham, Massachusetts.
85. Heckel, R., and Wagner, A. 1995. Ensuring Consistency of Conditional Graph Rewriting – a Constructive Approach. In Proceedings of the Joint COMPUGRAPH/SEMAGRAPH Workshop on Graph Rewriting and Computation (SEGRAGRA'95).
86. Hirayama, K., and Toyoda, J. 1995. Forming Coalitions for Breaking Deadlocks. In Proceedings of the First International Conference on Multi-Agent Systems (ICMAS-95), 155–162.
87. Hoffmann, J. 2000. A Heuristic for Domain Independent Planning and its Use in an Enforced Hill-climbing Algorithm. In Proceedings of the Twelfth International Symposium on Methodologies for Intelligent Systems.
88. Hooker, J.; Ottosson, G.; Thorsteinsson, E. S.; and Kim, H.-J. 1999. A Scheme for Unifying Optimization and Constraint Satisfaction Methods. *Knowledge Engineering Review*, to appear.

89. ILOG, Inc. 2000. Optimization Technology White Paper – A comparative study of optimization technologies. White Paper, ILOG, Inc., Mountain View, CA.
90. Isbister, K. 1995. Perceived Intelligence and the Design of Computer Characters. Lifelike Computer Characters (LCC'95), Snowbird, Utah.
91. Jacopin, É., and Penon, J. 2000. On the Path from Classical Planning to Arithmetic Constraint Satisfaction. In Papers from the AAAI-2000 Workshop on Constraints and AI Planning, Technical Report, WS-00-02, 18–24. AAAI Press, Menlo Park, California.
92. Joslin, D. 1996. Passive and Active Decision Postponement in Plan Generation. PhD thesis, University of Pittsburgh, Pittsburgh, PA.
93. Joslin, D. E., and Clements, D. P. 1999. Squeaky Wheel Optimization. *Journal of Artificial Intelligence Research* 10, 353–373.
94. Junker, U. 2000. Preference-based Search for Scheduling. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 904–909.
95. Jussien, N., and Lhomme, O. 2000. Local search with constraint propagation and conflict-based heuristics. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 169–174.
96. Kakas, A., and Miller, R. 1997. A Simple Declarative Language for Describing Narratives with Actions. *The Journal of Logic Programming* 31: 157–200.
97. Karmarkar, N. 1984. A New Polynomial-Time Algorithm for Linear Programming. *Combinatorica* 4: 373–395.
98. Kautz, H., and Selman, B. 1992. Planning as Satisfiability. In Proceedings of the Tenth European Conference on Artificial Intelligence (ECAI-92), 359–363.
99. Kautz, H., and Selman, B. 1996. Pushing the Envelope: Planning, Propositional Logic, and Stochastic Search. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 1194–1201.
100. Kautz, H., and Selman, B. 1998. BLACKBOX: A New Approach to the Application of Theorem Proving to Problem Solving. In Working Notes of the AIPS-98 Workshop on Planning as Combinatorial Search, 58–60.
101. Kautz, H., and Walser, J. P. 1999. State-space Planning by Integer Optimization. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 526–533.
102. Kirkpatrick, S.; Gelatt, C. D.; and Vecchi, M. P. 1983. Optimization by Simulated Annealing. *Science* 220(4598): 671–680.
103. Koehler, J. 1998. Planning under Resource Constraints. In Proceedings of the Thirteenth European Conference on Artificial Intelligence (ECAI-98), 489–493.
104. Koenig, S., and Liu, Y. 2000. Representations of Decision-Theoretic Planning Tasks. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), 187–195.
105. Kondrak, G., and van Beek, P. 1997. A Theoretical Evaluation of Selected Backtracking Algorithms. *Artificial Intelligence* 89: 365–387.
106. Korf, R. E. 2000. Recent Progress in the Design and Analysis of Admissible Heuristic Functions. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 1165–1170.
107. Knoblock, C. A. 1995. Planning, Executing, Sensing, and Replanning for Information Gathering. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), 1686–1693.
108. Kushmerick, N.; Hanks, S.; and Weld, D. 1995. An Algorithm for Probabilistic Planning. *Artificial Intelligence* 76: 239–286.

109. Kwok, C. T., and Weld, D. S. 1996. Planning to Gather Information. In Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96), 32–39.
110. Laborie, P., and Ghallab, M. 1995. Planning with Sharable Resource Constraints. In Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95), 1643–1649.
111. Le Pape, C. 1994. Implementation of Resource Constraints in ILOG Schedule: A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering* 3(2): 55–66.
112. Lieber, J., and Napoli, A. 1996. Using Classification in Case-Based Planning. In Proceedings of the Twelfth European Conference on Artificial Intelligence (ECAI-96), 132–137.
113. Malik, J., and Binford, T. O. 1983. Reasoning in Time and Space. In Proceedings of the Eighth International Joint Conference on Artificial Intelligence (IJCAI-83), 343–345.
114. Martin, D. L.; Cheyer, A. J.; and Moran, D. B. 1999. The open agent architecture: A framework for building distributed software systems. *Applied Artificial Intelligence* 13: 91–128.
115. Mattsson, C. 2000. The Tolkien Monster Encyclopedia. `http://home7.swipnet.se/~w-70531/Tolkien/`
116. McAllester, D; Selman, B.; and Kautz, H. 1997. Evidence for Invariants in Local Search. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 321–326.
117. Meyer auf'm Hofe, H. 1998. Finding Regions for Local Repair in Partial Constraint Satisfaction. In Proceedings of the Twentysecond Annual German Conference on Artificial Intelligence (KI-98).
118. McCarthy, J., and Hayes, P. J. 1969. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In Meltzer, B., and Mitchie, D. (eds.), *Machine Intelligence 4*, Edinburgh University Press.
119. Milano, M.; Ottosson, G.; Refalo, P.; and Thorsteinsson, E. S. 2000. The Benefits of Global Constraints for the Integration of Constraint Programming and Integer Programming. In Working Notes of the AAAI-2000 Workshop on Integration of AI and OR Techniques for Combinatorial Optimization.
120. Minton, S.; Bresina, J; and Drummond, M. 1994. Total-Order and Partial-Order Planning: A Comparative Analysis. *Journal of Artificial Intelligence Research* 2: 227–262.
121. Minton, S.; Johnston, M. D.; Philips, A. B.; and Laird, P. 1992 . Minimizing Conflicts: a Heuristic Repair Method for Constraint Satisfaction and Scheduling Problems. *Artificial Intelligence* 58: 161–205.
122. Mittal, S., and Falkenhainer, B. 1990. Dynamic Constraint Satisfaction Problems. In Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90), 25–32.
123. Mohr, R., and Henderson, T. C. 1986. Arc and Path Consistency Revisited. *Artificial Intelligence* 28(2): 225–233.
124. Muscettola, N. 1994. HSTS: Integrating Planning and Scheduling. In Zweben, M., and Fox, M. S. (eds.), *Intelligent Scheduling*, Morgan Kaufmann, 169–212.
125. Muslea, I. 1998. A General-Purpose AI Planning System Based on the Genetic Programming Paradigm. In Proceedings of the World Automation Congress (WAC'98).
126. Myers, K. L. 1999. CPEF – A Continuous Planning and Execution Framework. *AI Magazine* 20(4): 63–69.

127. Nareyek, A. 1997. Constraint-based Agents. In Papers from the 1997 AAAI Workshop on Constraints & Agents, Technical Report, WS-97-05, 45–50. AAAI Press, Menlo Park, California.
128. Nareyek, A. 1998. A Planning Model for Agents in Dynamic and Uncertain Real-Time Environments. In Proceedings of the Workshop on Integrating Planning, Scheduling and Execution in Dynamic and Uncertain Environments at the Fourth International Conference on Artificial Intelligence Planning Systems (AIPS'98), Technical Report, WS-98-02, 7–14. AAAI Press, Menlo Park, California.
129. Nareyek, A. 1998. Constraint-basierte Planung für Agenten in Computerspielen. In Proceedings of the Workshop on Deklarative KI-Methoden zur Implementierung und Nutzung von Systemen in Netzen at the 22. Jahrestagung Künstliche Intelligenz (KI-98), 21–30.
130. Nareyek, A. 1999. Structural Constraint Satisfaction. In Papers from the 1999 AAAI Workshop on Configuration, Technical Report, WS-99-05, 76–82. AAAI Press, Menlo Park, California.
131. Nareyek, A. 1999. Applying Local Search to Structural Constraint Satisfaction. In Proceedings of the IJCAI-99 Workshop on Intelligent Workflow and Process Management: The New Frontier for AI in Business.
132. Nareyek, A. 2000. AI Planning in a Constraint Programming Framework. In Hommel, G. (ed.), *Communication-Based Systems*, Kluwer Academic Publishers, 163–178.
133. Nareyek, A. 2000. Open World Planning as SCSP. In Papers from the AAAI-2000 Workshop on Constraints and AI Planning, Technical Report, WS-00-02, 35–46. AAAI Press, Menlo Park, California.
134. Nareyek, A. 2000. Intelligent Agents for Computer Games. In Proceedings of the Second International Conference on Computers and Games (CG 2000), to appear.
135. Nareyek, A. 2000. Constraint Programming for Computer Games: Mastering "Real"-World Requirements. In Proceedings of the Thirteenth International Conference on Applications of Prolog (INAP 2000).
136. Nareyek, A. 2001. Using Global Constraints for Local Search. In Freuder, E. C., and Wallace, R. J. (eds.), *Constraint Programming and Large Scale Discrete Optimization*, American Mathematical Society Publications, DIMACS Volume 57.
137. Nareyek, A., and Geske, U. 1996. Efficient Representation of Relations over Linear Constraints. In Proceedings of the Workshop on Constraint Programming Applications at the Second International Conference on Principles and Practice of Constraint Programming (CP96), 55–63.
138. Nau, D.; Cao, Y.; Lotem, A.; and Muñoz-Avila, H. 1999. SHOP: Simple Hierarchical Ordered Planner. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), 968–973.
139. Nemhauser, G. L. and Wolsey, L. A. 1988. *Integer and Combinatorial Optimization.* Reading, John Wiley & Sons, Inc.
140. Nievergelt, J.; Gasser, R.; Maser, F.; and Wirth, C. 1995. All the Needles in a Haystack: Can Exhaustive Search Overcome Combinatorial Chaos? In van Leeuwen, J. (ed.), *Computer Science Today*, Springer LNCS, 254–274.
141. Nuijten, W. P. M. 1994. Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach. PhD Thesis, Eindhoven University of Technology, Eindhoven, The Netherlands.
142. Oddi, A., and Smith, S. 1997. Stochastic Procedures for Generating Feasible Schedules. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 308–314.

143. Paredis, J. 1999. Coevolutionary Algorithms. In Bäck, T.; Fogel, D.; and Michalewicz, Z. (eds.), *The Handbook of Evolutionary Computation*, 1st supplement, Oxford University Press.
144. Paolucci, M.; Kalp, D.; Pannu, A.; Shehory, O.; and Sycara, K. 1999. A Planning Component for RETSINA Agents. In Wooldridge, M., and Lesperance, Y. (eds.), *Intelligent Agents VI*, Springer LNAI.
145. Pell, B.; Bernard, D. E.; Chien, S. A.; Gat, E.; Muscettola, N.; Nayak, P. P.; Wagner, M. D.; and Williams, B. C. 1996. A Remote Agent Prototype for Spacecraft Autonomy. In Proceedings of the SPIE Conference on Optical Science, Engineering, and Instrumentation.
146. Penberthy, J. S., and Weld, D. S. 1992. UCPOP: A Sound, Complete, Partial Order Planner for ADL. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), 102–114.
147. Penberthy, J. S., and Weld, D. S. 1994. Temporal Planning with Continuous Change. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 1010–1015.
148. Peot, M., and Smith, D. 1992. Conditional Nonlinear Planning. In Proceedings of the First International Conference on AI Planning Systems, 189–197.
149. Pesant, G., and Gendreau, M. 1999. A Constraint Programming Framework for Local Search Methods. *Journal of Heuristics* 5(3): 255–279.
150. Prosser, P. 1993. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence* 9(3): 268–299.
151. Pryor, L., and Collins, G. 1996. Planning for Contingencies: A Decision-based Approach. *Journal of Artificial Intelligence Research* 4: 287–339.
152. Puget, J.-F., and Leconte, M. 1995. Beyond the Glass Box: Constraints as Objects. In Proceedings of the 1995 International Logic Programming Symposium (ILPS'95), 513–527.
153. Rabideau, G.; Chien, S.; Willis, J.; and Mann, T. 1999. Using Iterative Repair to Automate Planning and Scheduling of Shuttle Payload Operations. In Proceedings of the Eleventh Annual Conference on Innovative Applications of Artificial Intelligence (IAAI-99), 813–820.
154. Rabideau, G.; Knight, R.; Chien, S.; Fukunaga, A.; and Govindjee, A. 1999. Iterative Repair Planning for Spacecraft Operations in the ASPEN System. International Symposium on Artificial Intelligence Robotics and Automation in Space (iSAIRAS 99).
155. Régin, J.-C. 1998. Minimization of the Number of Breaks in Sports Scheduling Problems using Constraint Programming. In Proceedings of the DIMACS Workshop on Constraint Programming and Large Scale Discrete Optimization, P7: 1–23.
156. Reticular Systems, Inc. 1999. AgentBuilder – An Integrated Toolkit for Constructing Intelligent Software Agents. White Paper, Revision 1.3. Reticular Systems, Inc., Dan Diego, CA.
157. Rintanen, J., and Jungholt, H. 1999. Numeric State Variables in Constraint-based Planning. In Proceedings of the Fifth European Conference on Planning (ECP-99).
158. Rozenberg, G. ed. 1997. *The Handbook of Graph Grammars. Volume I: Foundations.* Reading, World Scientific.
159. Sabin, D., and Freuder, E. C. 1996. Configuration as Composite Constraint Satisfaction. In Proceedings of the Artificial Intelligence and Manufacturing Research Planning Workshop, 153–161.
160. Sacerdoti, E. D. 1974. Planning in a Hierarchy of Abstraction Spaces. *Artificial Intelligence* 5(2): 115–135.

161. Sacerdoti, E. D. 1975. The Nonlinear Nature of Plans. In Proceedings of the Fourth International Joint Conference on Artificial Intelligence (IJCAI-75), 206–214.
162. Sadeh, N., and Fox, M. 1996. Variable and Value Ordering Heuristics for the Job Shop Scheduling Constraint Satisfaction Problem. *Artificial Intelligence* 86: 1–41.
163. Schrag, R; Boddy, M; and Carciofini, J. 1992. Managing Disjunction for Practical Temporal Reasoning. In Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning (KR'92), 36–46.
164. Schuurmans, D., and Southey, F. 2000. Local search characteristics of incomplete SAT procedures. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 297–302.
165. Selman, B.; Levesque, H.; and Mitchell, D. 1992. A New Method for Solving Hard Satisfiability Problems. In Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92), 440–446.
166. Selman, B.; Kautz, H.; and Cohen, B. 1996. Local Search Strategies for Satisfiability Testing. In Johnson, D. S., and Trick, M. A. (eds.), *Cliques, Coloring, and Satisfiability*, DIMACS Volume 26: 521–532.
167. Smith, S. F. 1994. OPIS: A Methodology and Architecture for Reactive Scheduling. In Zweben, M., and Fox, M. S. (eds.), *Intelligent Scheduling*, Morgan Kaufmann, 29–66.
168. Sqalli, M. H.; Purvis, L.; and Freuder, E. C. 1999. Survey of Applications Integrating Constraint Satisfaction and Case-Based Reasoning. In Proceedings of the First International Conference and Exhibition on the Practical Application of Constraint Technologies and Logic Programming (PACLP99).
169. Srivastava, B., and Kambhampati, S. 1999. Scaling up Planning by teasing out Resource Scheduling. In Proceedings of the Fifth European Conference on Planning (ECP-99).
170. Stern, A.; Frank, A.; and Resner, B. 1998. Virtual Petz: A Hybrid Approach to Creating Autonomous Lifelike Dogz and Catz. In Proceedings of the Second International Conference on Autonomous Agents (AGENTS98), 334–335.
171. Stern, A. 1999. AI Beyond Computer Games. 1999 AAAI Symposium on Computer Games and Artificial Intelligence.
172. Tate, A. 1977. Generating Project Networks. In Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77), 888–893.
173. Tumer, K., and Wolpert, D. 2000. Collective Intelligence and Braess' Paradox. In Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-2000), 104–109.
174. Vaessens, R. J. M.; Aarts, E. H. L.; and Lenstra, J. K. 1994. Job Shop Scheduling by Local Search. Technical Report, COSOR Memorandum 94-05, Eindhoven University of Technology, Department of Mathematics and Computing Science.
175. Van Beek, P., and Chen, X. 1999. CPlan: A Constraint Programming Approach to Planning. In Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99), 585–590.
176. Van Lent, M., and Laird, J. 1999. Developing an Artificial Intelligence Engine. In Proceedings of the 1999 Game Developers Conference (GDC 1999), 577–588.
177. Veloso, M.; Carbonell, J.; Pérez, A.; Borrajo, D.; Fink, E.; and Blythe, J. 1995. Integrating Planning and Learning: The PRODIGY Architecture. *Journal of Experimental and Theoretical Artificial Intelligence* 7(1).

178. Veloso, M.; Muñoz-Avila, H.; and Bergmann, R. 1996. Case-based Planning: Selected Methods and Systems. *AI Communications* 9(3): 128–137.
179. Verfaillie, G., and Schiex, T. 1994. Solution Reuse in Dynamic Constraint Satisfaction Problems. In Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94), 307–312.
180. Vossen, T.; Ball, M.; Lotem, A.; and Nau, D. 1999. On the Use of Integer Programming Models in AI Planning. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), 304–309.
181. Voudouris, C., and Tsang, E. 1995. Guided Local Search. Technical Report CSM-247, University of Essex, Department of Computer Science, Colchester, United Kingdom.
182. Waldinger, R. 1977. Achieving Several Goals Simultaneously. In Elcock, E., and Michie, D. (eds.), *Machine Intelligence 8*, Ellis Horwood Limited, 94–136.
183. Wallace, R. J., and Freuder, E. C. 1996. Anytime Algorithms for Constraint Satisfaction and SAT problems. *SIGART Bulletin* 7(2).
184. Wallace, R. J., and Freuder, E. C. 2000. Dispatchable execution of schedules involving consumable resources. In Proceedings of the Fifth International Conference on Artificial Intelligence Planning and Scheduling (AIPS-2000), 283–290.
185. Walser, J. P. 1997. Solving Linear Pseudo-Boolean Constraint Problems with Local Search. In Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), 269–274.
186. Warren, D. H. D. 1974. WARPLAN: A System for Generating Plans. Department of Computational Logic Memo 76, University of Edinburgh, Scotland.
187. Warren, D. H. D. 1976. Generating Conditional Plans and Programs. In Proceedings of the Summer Conference on Artificial Intelligence and Simulation on Behavior, 344–354.
188. Weizenbaum, J. 1966. ELIZA – A Computer Program for the Study of Natural Language Communication between Man and Machine. *Communications of the ACM* 9(1): 36–45.
189. Wilkins, D. E.; Myers, K. L.; Lowrance, J. D.; and Wesley, L. P. 1995. Planning and Reacting in Uncertain and Dynamic Environments. *Journal of Experimental and Theoretical AI* 7(1): 197–227.
190. Williamson, M., and Hanks, S. 1994. Optimal Planning with a Goal-Directed Utility Model. In Proceedings of the Second International Conference on AI Planning Systems (AIPS-94), 176–181.
191. Wolfman, S. A., and Weld, D. S. 1999. The LPSAT Engine & its Application to Resource Planning. In Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99), 310–316.
192. Woodcock, S. 2000. Game AI: The State of the Industry. *Game Developer*, August 2000.
193. Wooldridge, M., and Jennings, N. R. 1995. Intelligent Agents: Theory and Practice. *The Knowledge Engineering Review* 10(2): 115–152.
194. Wright, I., and Marshall, J. 2000. More AI in Less Processor Time: 'Egocentric' AI. *Gamasutra*, June 2000.
`http://www.gamasutra.com/features/20000619/wright_01.htm`
195. Zilberstein, S. 1996. Using Anytime Algorithms in Intelligent Systems. *AI Magazine* 17(3): 73–83.
196. Zweben, M.; Daun, B.; Davis, E.; and Deale, M. 1994. Scheduling and Rescheduling with Iterative Repair. In Zweben, M., and Fox, M. S. (eds.), *Intelligent Scheduling*, Morgan Kaufmann, 241–255.
197. Zwick, R., and Lee, C. C. 1999. Bargaining and search: An experimental study. *Group Decision and Negotiation* 8(6), 463–487.

Index